12-2017

SystemVerilog Verification of Wishbone-Compliant Serial Peripheral Interface

Avinash Srinivasan
as7409@rit.edu

SystemVerilog Verification Of Wishbone-Compliant Serial Peripheral Interface

by

Avinash Srinivasan

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
December, 2017

I dedicate my work to my mentor Professor Mark Indovina, my father Srinivasan Krishnarao, mother Ratna Srinivasan, brother Abhishek, sister-in-law Swathi and friends for their support during my thesis.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper is original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This research project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Avinash Srinivasan
December 19, 2017

Acknowledgements

I wholeheartedly thank my graduate advisor and mentor Professor Mark Indovina for his continued support, feedback, encouragement and guidance through out my graduate career and research. I also thank Dr. Dorin Patru and Dr. Jayanthi Venkataraman for their support and guidance through out my graduate career.

Abstract

Synchronous serial interfaces provide economical on-board communication between the processor, digital to analog and analog to digital converters, memory, and other building blocks on the chip. A number of **I**ntegrated **C**ircuit (**IC**) manufacturers develop and produce components that are compatible with serial interfaces. The common serial interfaces include **S**erial **P**eripheral **I**nterface (**SPI**), **I**nter-**I**ntegrated **C**ircuit (**I^{2}C**), **U**niversal **A**synchronous **R**eceiver **T**ransmitter (**UART**), and **U**niversal **S**erial **B**us (**USB**). SPI is widely used and advantageous over other serial interfaces due to its features of simplicity, low cost, clock synchronous, and non-interrupting high-speed data transfer rate. SPI is the core controller of the design. An open source hardware computer bus Wishbone is selected as the host controller enabling parallel data exchange for faster communication. Both the hardware buses employ a master-slave configuration which makes the bus-interfacing easier.

This research presents a verification environment using SystemVerilog for the SPI Master device. An existing design from Open Cores is re-used, described as per latest specifications in Verilog at the **R**egister **T**ransfer **L**evel (RTL) and is in conformity with the design-reuse methodology. This paper provides an understanding of the verification, layered test benches, **O**bject-**O**riented **T**echnology (**OOT**), SystemVerilog, SPI features, SPI advantages, disadvantages, and applications, SPI data transmission and transfer formats, SPI registers, SPI sub-system and Wishbone-SPI architecture, and the test bench development methodology. The focus is to understand how OOT and SystemVerilog improve productivity and functional coverage in a verification environment by the use of different constructs, constrained-random techniques, coverage, and assertions. A test bench was developed to verify the SPI master core. Testbench components include a random transaction generator, a Wishbone driver, an SPI master as the design under test, a receiver as the SPI slave, a monitor with tasks to monitor the host and the core, test cases, and a scoreboard to record metrics, assertions and store expected and actual data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An **A**pplication **S**pecific **I**ntegrated **C**ircuit (**ASIC**) is is a special purpose device designed for a specific application and perform a specific function. A modern ASIC may include microprocessors or microcontrollers, memory and relevant building blocks on a single chip called a System-**on**-Chip (**SoC**). The presence of a microprocessor or microcontroller distinguishes an ASIC as an SoC or a non-SoC. A mobile phone is a classic example of an embedded system which embeds a SoC. The mobile phone SoC includes a Microcontroller, a **D**igital **S**ignal **P**rocessor (**DSP**), Memory, Peripherals, **D**irect **M**emory **A**ccess **c**ontroller (**DMA** controller), **L**iquid **C**rystal **D**isplay controller (**LCD** controller) and a battery. An SoC is always developed to provide better reliability, lower cost, lower area, lower power, better performance and to meet the technology advancements and product end-user requirements. An Embedded System or a SoC uses a microcontroller or a microprocessor for intelligent control mechanism, display drivers for displaying the content, memory for storage, data converters, remote input and output ports, and integrated circuits for communication interfaces. The desired functionality is achieved when multiple components of the chip operate, communicate and collaborate with each other. These components interact with each other through communication controllers or interfaces. The interfaces may be serial or parallel depending on the transmission of data and are selected based on the protocols supported by the components. A few interfaces include **S**erial **P**eripheral **I**nterface (**SPI**), **I**nter-**I**ntegrated **C**ircuit (**I^{2}C**), **U**niversal **A**synchronous **R**eceiver **T**ransmitter (**UART**), **U**niversal **S**erial **B**us (**USB**), Ethernet, Bluetooth, WiFi, **A**vionics **F**ull **D**uple**X** Switched Ethernet (**AFDX**), **C**ontroller **A**rea **N**etwork (**CAN),** **A**eronautical **R**adio **INC** 429 (**ARINC 429**), **MIL**itary-**ST**andar**D**-1553 (**MIL-STD-1553**). SPI, I^{2}C, UART, and USB are most common serial interfaces used.

SPI is advantageous over I^{2}C and UART as the data exchange is synchronous to a clock signal and is configured as a master-slave device. SPI master device is the core controller. An open source hardware computer bus Wishbone is the host controller to SPI and it enables parallel exchange of data with the processor. It is configured as a master-slave like the SPI, which makes the interfacing easier. As a result, the core controller is a slave of the host controller. Motorola developed the SPI communication controller. It is a data-link and a De facto standard. Parallel data from the microprocessor or microcontroller is transmitted through Wishbone to the SPI Master device. This data is serialized in bits with one bit being transmitted to the slave in a clock cycle. The slave acknowledges the master in a similar way. The transmission and reception of data between the SPI master device and the SPI slave device happen simultaneously and the communication is full duplex. SPI is a 3+-wire (typically 4), synchronous serial controller. The interfaces of SPI include **S**erial **C**loc**K** (**SCK**), **M**aster **I**n **S**lave **O**ut (**MISO**)**,** **M**aster **O**ut **S**lave **I**n (**MOSI**) and an optional **S**lave **S**elect (**SS**). Unlike Wishbone, SPI supports only a single master and a multiple slave configuration. At any given instance, only the master device can initiate a data frame transfer with only one slave device by generating the clock signal through SCK and selecting the slave device through SS. The SPI master operates as an 8-bit shift register. The SPI slave operates as an 8-bit shift register. The SPI master and slave devices are tied together through the interfaces of the SPI. Ideally, the **L**east **S**ignificant **B**it **(LSB)** of the master is transferred into the **M**ost **S**ignificant **B**it **(MSB)** of the slave through MOSI data line and the **M**ost **S**ignificant **B**it **(MSB)** of the slave is transferred into the **L**east **S**ignificant **B**it **(LSB)** of the master through the MISO data line. This two-way communication works similar to a 16-bit circular shift register. **SCK** synchronizes the sampling and shifting of data on the MISO and MOSI by assigning the clock phase, clock polarity, and clock frequency.

SPI is widely used in the industry for its swift data transfer rates with a bit rate of half the clock rate and can operate at a speed of up to 1.1 Mbps. A flexible design methodology that is used with semiconductor IP cores is the Wishbone Bus Interconnection. It enables faster design re-usability in the integration of the chip due to its simple, compact, and logical IP core hardware interfaces requiring less number of logic gates. It enables parallel data communication on a chip and is selected as the host controller to the SPI.

The design implementation of these hardware platforms can be carried out using **Verilog** **H**ardware **D**escription **L**anguage (**Verilog HDL**) or **V**ery High-Speed Integrated Circuit **H**ardware **D**escription **L**anguage **(VHDL**). But Verilog or VHDL is not enough for a complete verification of these platforms. The traditional approach of verification is to functionally simulate the hardware, develop and emulate the software by transacting all the input combinations to the design under test and monitoring the expected behavior. This approach is time-consuming and poses a major threat to design and verification re-work. An automated system-level verification with re-usable and scalable components is required to verify complex designs. A majority of the **E**lectronic **D**esign **A**utomation (**EDA)** tool vendors use automated verification methodologies and **O**bject-**O**riented **T**echnology (**OOT**). The automated verification methodologies include SystemVerilog, **U**niversal **V**erification **M**ethodology (**UVM**), **V**erification **M**ethodology **M**anual (**VMM)**, **A**dvanced **V**erification **M**anual (**AVM**), and **O**pen **V**erification **M**anual (**OVM**) [6]. SystemVerilog is extensively used in verifying complex SoC designs due to higher level abstractions and use of OOT. It combines specifications, design, simulation, verification and validation into a single language by improving the communication between design and verification engineers aiding an efficient design and verification process. Three-fourths of the time in the product development life cycle is spent on verification and testing. Given adequate time and resources, direct testing is used to exploit features of verification environment. The increasing levels of design complexity force the usage of random tests by automation in addition to certain directed tests to cover all the input combinations and achieve a 100% functional and code coverage metrics. Unlike traditional directed testing, random tests start with the expected behavior and work backward to check the outputs with automatically generated input stimulus through transaction generators. A verification environment is developed through a layered approach. The layered approach divides the tasks into Signal, Functional, Command, Scenario and Test layers developed independently. This environment is built upon a verification plan which describes the aspects of the design to be verified, the techniques to be used and the creation of constrained random tests. Code and Functional coverage metrics are captured to check all aspects of the verification plan. Functional coverage measures the test success rate to evaluate the performance. Code Coverage measures the coding standards metrics.

A configurable and reusable verification environment to validate the Wishbone-compliant SPI core controller is developed using SystemVerilog. The test scenarios include different character lengths, reset condition, data transmission modes, interrupt checks and Auto Slave Select function. The constrained-random stimulus generator generates the transactions and the driver is in the form of Wishbone. The design under test is the SPI master. The receiver is the SPI slave. Monitors are used on the host-side and core-side to check the actual data and the expected data. A scoreboard is used for temporary storage, and to capture the coverage metrics and assertions.

1.1 Research Goals

The goal is to build a configurable and re-usable complex verification test bench that validates the Wishbone-compliant SPI communication controller. A top-down design [7] and a layered verification approach are followed to achieve this goal. The objectives include understanding the system requirements and specifications, Wishbone-SPI architecture, SPI registers, SPI data transmission modes, the creation of a testbench environment, and analysis of the data transmission.

1.2 Contributions

The major contributions to the Wishbone-SPI Core communication controller include:

- Understanding the SPI sub-system architecture, Wishbone-SPI architecture, verification, Object-oriented Technology and SystemVerilog.
- Modifying existing designs of the Wishbone-SPI communication controller as per the latest specifications.
- Developing the testbench components of the verification environment using SystemVerilog constructs, random stimulus, assertions, and coverage.
- Connecting the components of the verification environment to verify the transmission of data for different character lengths and data transfer formats.

1.3 Organization

This paper has been organized in the following way:

- Chapter 2 focuses verification, strategies involved in creating a verification environment, and benefits of layered approach.
- Chapter 3 describes the benefits of using object-oriented technology and SystemVerilog.

- Chapter 4 describes in detail the Serial Peripheral Interface, its salient features, advantages, disadvantages, applications, data transmission, registers, Wishbone, SPI sub-system, and Wishbone-SPI architecture.
- Chapter 5 outlines the developed verification environment, the testbench components, the testbench code methodology, and the testbench results.
- Chapter 6 concludes the paper by highlighting the possible scope of the possible future work in this research.

Chapter 2

Verification

2.1 Overview

The complexity of ASIC and SoC designs increases exponentially [8]. Verification of these hardware platforms is highly essential as a bug in silicon can lead to increased costs. Enhanced verification strategies using directed and constrained random tests are required to ensure successful design implementations. The process of demonstrating a design's functional correctness is verification. The ultimate goal of verification is to minimize the costs, and achieve maximum code and functional coverage in a short span of time. Verification determines incorrect specifications, misinterpretations, incorrect interactions between cores and identifying unexpected system behavior through strategies like functional verification, object-oriented analysis, assertions, and randomization. A verification environment can be developed methodically by addressing concerns as to what aspects of the design has to be verified and how should the design be verified. Fig. 2.1 shows the different strategies involved in verification.

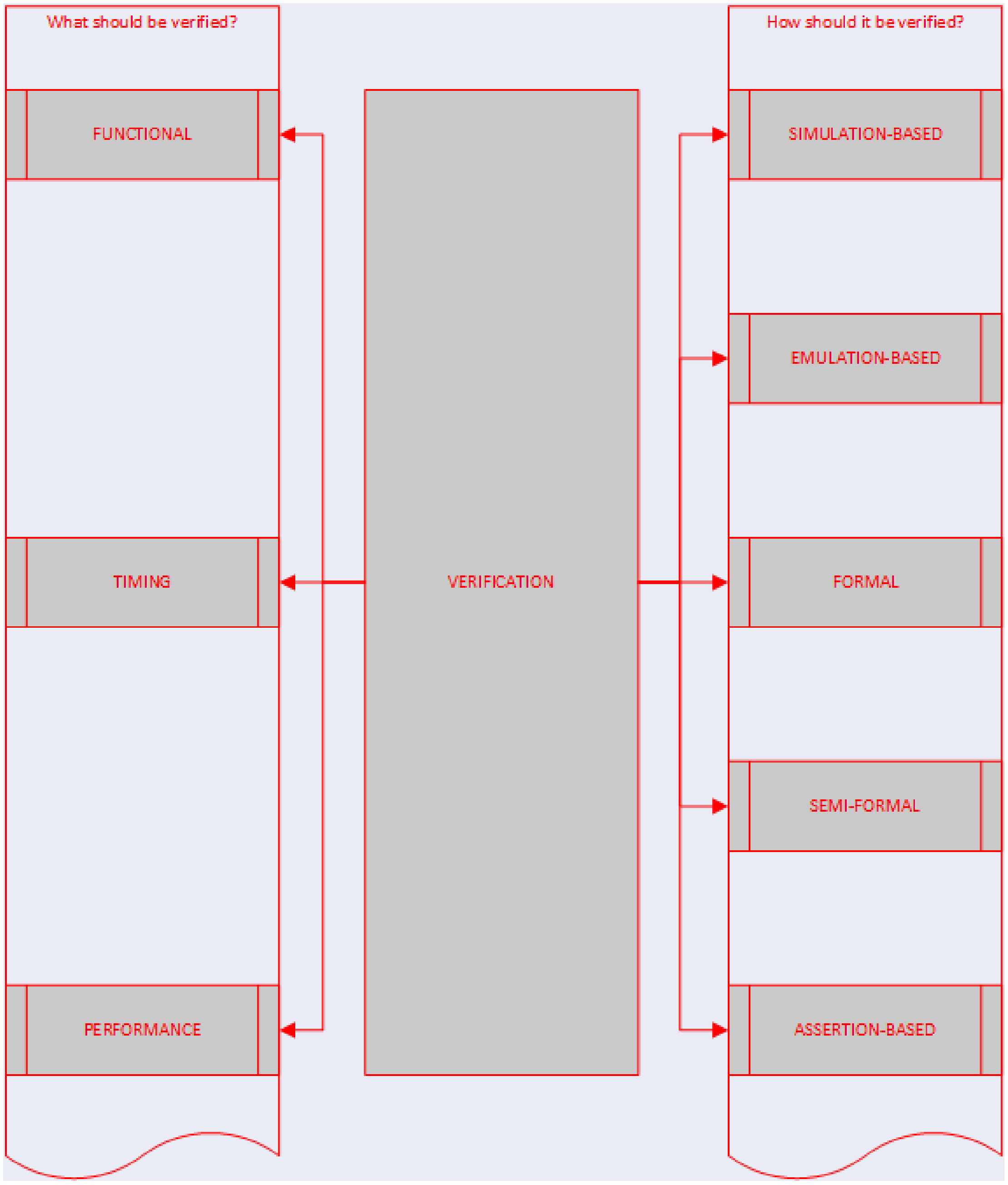


Figure 2.1: Verification Strategies

2.1.1 Aspects

A design has certain logic for a specific application. The logic must be designed as per specifications which include design constraints, timing, performance issues, power and area constraints. The following verification strategies define the aspects of the design to be verified and are divided into functional, timing and performance.

2.1.1.1 Functional

It is a process of verifying the logic of the design under test that conforms to the specifications using a mathematical model. It is cumbersome due to the enormous test cases present to verify the logic. It cannot be the only source of verification as it takes a lot of time and effort. It is supported by logic simulation, formal verification, RTL verification, Gate and Netlist level verification and emulation.

2.1.1.2 Timing

It validates the path delays from the primary input to the primary output to ensure that these delays are neither too short nor too long. It validates the clock pulses to ensure that they are neither too wide nor too narrow and the pulse width is retained as required. This is a kind of an electrical rule check and uses static and dynamic timing analysis to verify the timing constraints.

2.1.1.3 Performance

It determines whether a design meets the specified performance criteria. It may be performed in a hierarchical manner. For example, individual cells of the integrated circuit may be tested against desired performance goals during the initial pass. Testing may then proceed at the block level and finally at the full chip level for subsequent passes to meet performance criterion.

2.1.2 Methods

The logic, timing, performance, and power constraints can be verified using simulations, emulations, formal methods, assertions, and co-verification of hardware and software. The following verification strategies support the above verification strategies and are selected based on the complexity of the design.

2.1.2.1 Simulation-based

It is the traditional approach which is still used to verify any design and is one of the major components of the verification space. It requires the generation of input sequences or vectors and derives the reference outputs. The input sequences generated can be directed inputs, random inputs and constrained random inputs. It is input-driven and requires the generation each and every data-point input to capture the system's behavior. As a result, it generally requires a longer simulation time, especially for larger designs. This verification has to be supported by other verification strategies to achieve maximum functional and code coverage.

2.1.2.2 Emulation-based

This technique needs a certain emulation system like an FPGA-based board. It can provide a shorter verification cycle of the design and cannot be the only source of verification. The time and logic required to design a specific application must also include a board development time.

2.1.2.3 Formal

Unlike the simulation-based technique, this verification starts by deciding the desired output behavior of the design under test. A formal checker approves or disproves the obtained results. It is output-driven and a collection of data points or properties generated by a random stimulus generator are verified. As a result, missing an input space does not occur. It is the most widely used strategy for modern day SoCs but the main drawback is the use of extensive memory and a longer run-time. It consists of two approaches namely model checking and formal equivalence checking [9].

2.1.2.4 Semi-Formal

A handful of tool vendors consider adopting this strategy in verifying a design as it combines the simulation-based and formal verification. It achieves better results, and coverage metrics with lesser verification time in the product development life cycle.

2.1.2.5 Assertion-based

An assertion is the description of the property of a design as per the specifications. Assertions help in validating the design's intended behavior. The benefits of an assertion-based verification technique include improving the error detection for spotting simulation errors, improving the ability to debug effectively, improving the observability, usage in formal verification and dynamic simulation-based and providing a correct functional coverage. Two types of assertions can be used and these include immediate and concurrent assertions [10][11].

2.1.2.6 Co-verification of Hardware and Software

It provides the facility of verifying the hardware and the software functionality of an application specific design simultaneously. It is required due to the complexity, and demand for low cost and high volume consumer products. It requires a shorter time of verifying the complete design. It verifies the execution of the embedded system software accurately on the target embedded system hardware before the design is committed to fabrication. The simulation of the hardware logic behaves as the real hardware but it is actually an application software on a workstation. The benefits include an early access to the hardware design for the software developers and presence of stimulus for hardware engineers [12, 13].

2.2 Verification Plan: A Layered Approach

The goal of verification is to ensure that the device functions accurately and effectively as per the specifications and the expected behavior [14]. A verification plan is created to explain different aspects of the design to be verified based on the specifications and the techniques that are to be used to achieve the goal. The traditional approach is to generate the stimulus, apply the stimulus to the design under test and capture the results, check for the correctness against specifications, and measure the progress. A flat testbench is developed based on the plan, which wraps around the design under test. Different scenarios are created in the testbench which provide stimulus to the design to capture the results. These results are checked with the expected results from the testbench. The testbench environment has different levels of abstraction, creates transactions and sequences which has to be later transformed to bit vectors [14]. As a result, modulating the flat testbench programmed in Verilog into different layers and smaller chunks of logic makes it easier, efficient and maintainable. As a result, a layered testbench is developed and divided into signal, command, functional and scenario layers [14].

The **Signal** Layer is the lowest level in the layered testbench. It holds the design and the interface signals that connect it to the testbench.

The **Command** Layer is the next level which holds the driver to drive input transactions to design, monitor the transactions through checkers and assertions. The driver drives the input transactions to the design from the stimulus generator through single commands. The design's output drives the monitor to group them into commands by considering signal transitions. Assertions validate the signals integrity by considering the changes in the signals and the changes across an entire command. Fig. 2.2 showcases the signal and command layers of the environment.

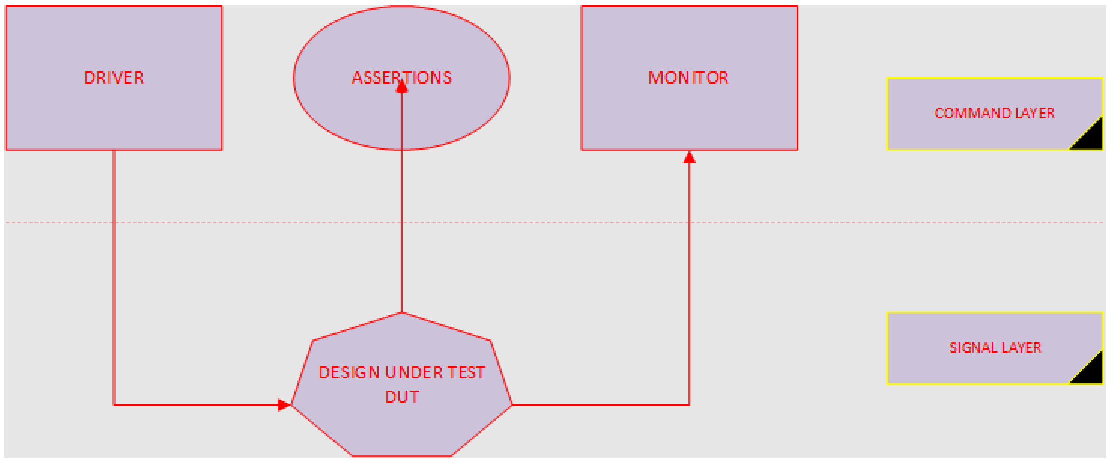


Figure 2.2: Signal and Command Layers

The **Functional** Layer forms the next layer and is one of the key layers of the layered testbench. It consists of the agent or stimulus, scoreboard, and checkers. The agent receives higher level transactions and splits them into individual transactions to be sent to the driver and scoreboard. Scoreboard predicts the results of the occurred transactions. Checker is used to comparing the commands received from the monitor with the predicted results from the scoreboard [14]. Fig. 2.3 showcases the environment with the functional layer included.

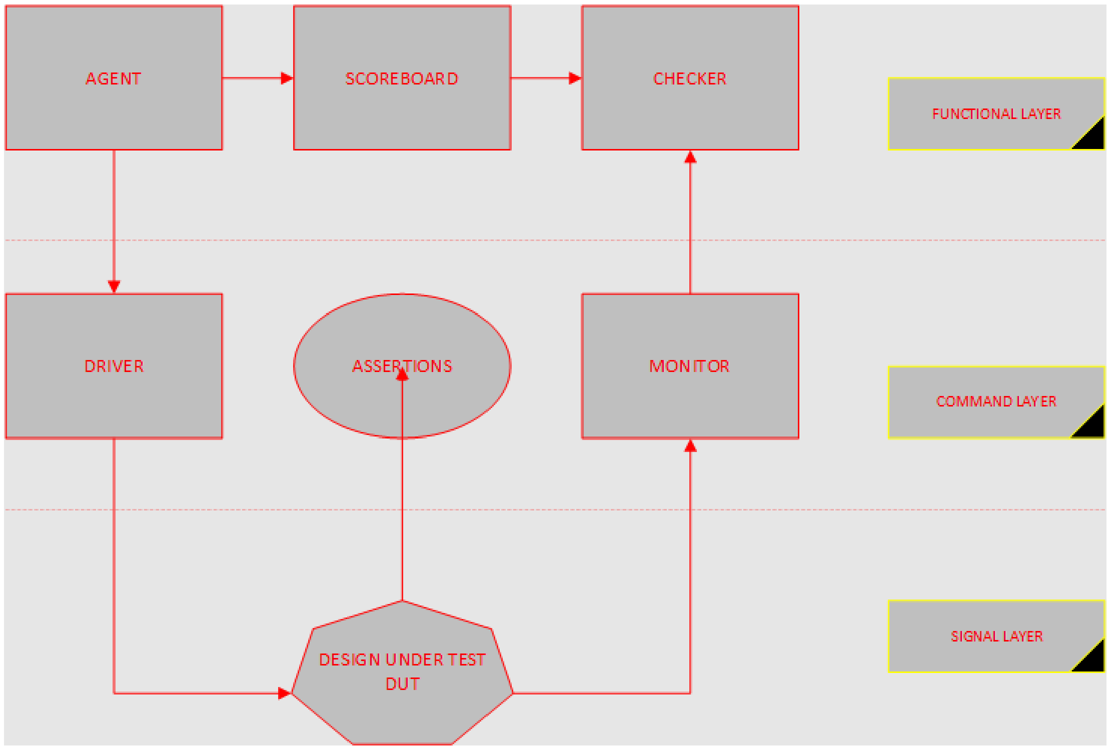


Figure 2.3: Functional, Signal and Command Layers

The **Scenario** Layer is primarily responsible for generating the test scenarios or test cases required to drive the Functional layer. Usually, constrained random values of different parameters are generated by the scenario layer and is responsible for working of the other layers or steps in this verification testbench [14].

The scenario, command and functional layers form the testbench environment which are depicted in Fig. 2.4.

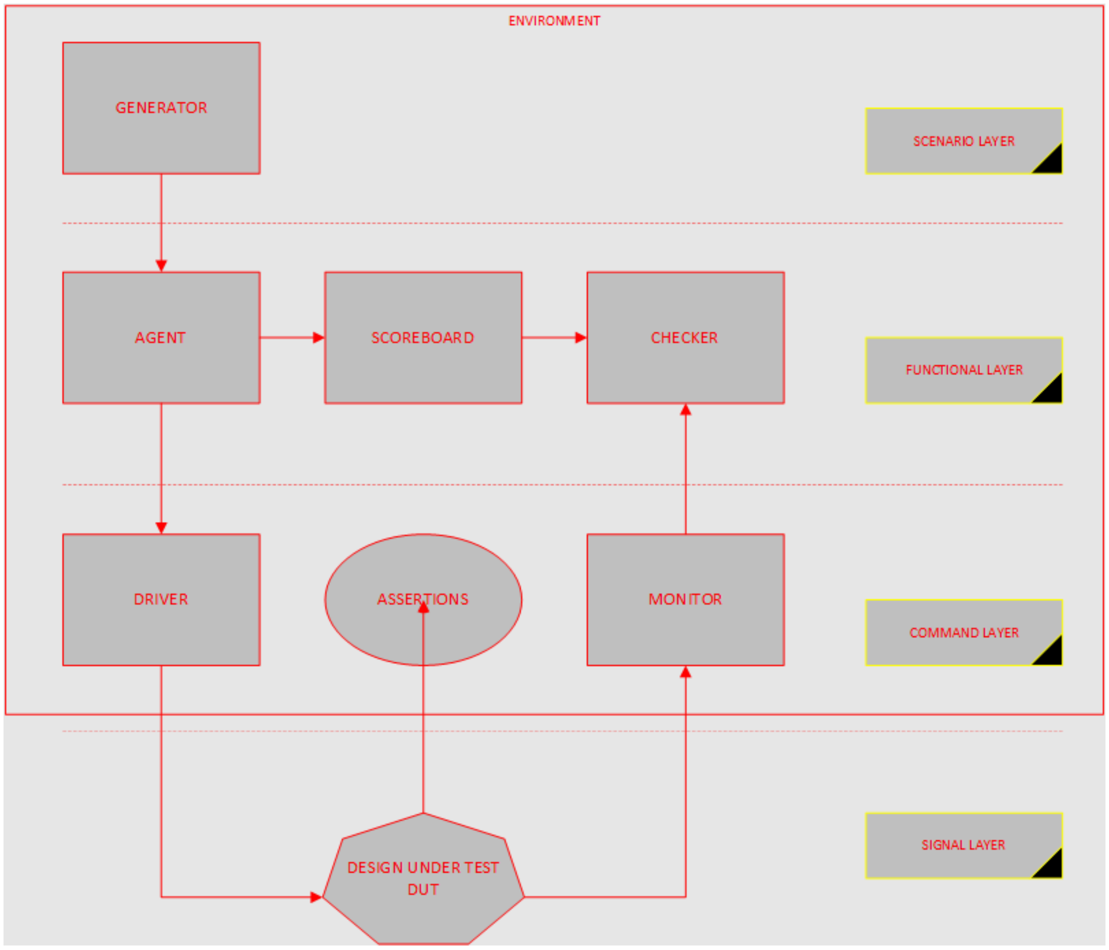


Figure 2.4: Scenario, Functional, Signal and Command Layers

The **Test** Layer and the **Functional Coverage** Layer form the top layers of a testbench environment. The top-level test only guides the efforts of other components of the testbench and it contains the constraints to create the stimulus. Functional Coverage Layer only measures the progress of all tests to check if all the verification plan requirements are met. Fig. 2.5 represents a complete Testbench Environment with all the layers included.

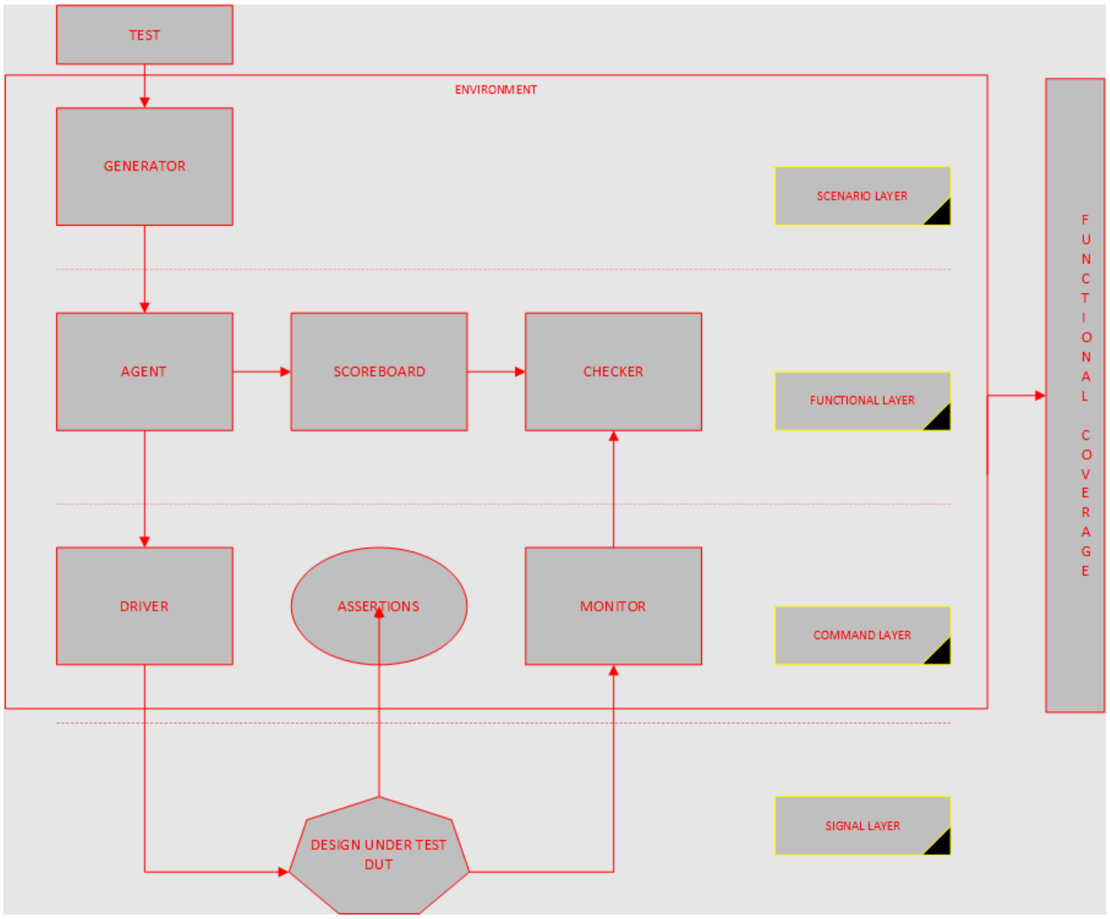


Figure 2.5: Complete Layered Testbench

2.3 Benefits of a Layered Approach

Benefits always follow necessity. In order to fulfill the drawbacks of a flat or traditional testbench, a layered testbench approach was formulated to achieve a 100% code and functional coverage by using directed and constrained random stimulus. The benefits of a layered testbench over a flat testbench in a verification environment are listed below:

- The focus has shifted from creating a stimulus to checking the expected system behavior.
- Extended support for the generation of constrained random stimulus.
- Support towards re-usability of verification environments.

- Improving the code and functional coverage through constrained random stimulus.
- Reducing the tool integration time.
- Allows creating a flexible, adaptable and a scalable testbench environment to increase the robustness of functional verification.

Chapter 3

SystemVerilog

3.1 Object-Oriented Technology

3.1.1 Overview

Object-**O**riented **T**echnology (**OOT**) is not a panacea and has been the predominant software paradigm for developing real-time embedded and application software. Its advantages of real-world modeling, reduced maintenance, flexibility and high-code re-usability implements a systems design without modifying the already existing changes. It allows for a consistency across different model views by providing a better reliability, better safety attributes and a better abstraction of the problem domain. Real-world entities or objects have four characteristics namely identity (instance), attributes (data), behavior (methods), and state (operating environment).

Increased complexity, technology advancement, speed and performance in memory concerns in a chip lead to an enormous amount of data transfer. A number of tasks are required to control or handle the data flow. This leads to application maintenance and behavioral problems, as behavior always follows the data. OOT eases in reducing the complexity by abstractly packaging self-sufficient modular pieces of the code, enabling better interaction or behavior between objects. It shifts the focus from objects to the behavior by modifying existing solutions to solve new problems. OOT supports code re-usability by using its features of Abstraction, Encapsulation, Inheritance, and Polymorphism[15].

3.1.1.1 Abstraction

Abstraction is hiding all but the relevant data of objects, to reduce complexity. Abstraction separates the interfaces from the design implementation. The data abstraction protects the internals of a class from user-level errors resulting in lesser modifications to the user-level code for changes in requirements specification over time [15].

3.1.1.2 Encapsulation

Encapsulation is combining and packaging all the related data, the related tasks, and the related functions together. The manipulation of data over time poses a threat to the misuse of data. Encapsulation ensures the safety of data and does not allow any outside interference to misuse the data.

3.1.1.3 Inheritance

Inheritance defines the hierarchies of related classes. A class having the properties and attributes of a parent-class can be defined as a derived class of the parent class. This ensures maintainability of an application and a shorter code. It ensures a faster implementation and execution of the application by promoting re-usability. It basically implements 'is a' relationship between the different classes of an application.

3.1.1.4 Polymorphism

Objects or real-world entities which encapsulate our understanding of the problem space [16]. The objects are created in a class and may have different forms or may be of different types. This ability of objects is Polymorphism. It occurs only if there is a defined hierarchy of classes. Two classes can have functions or tasks with the same name and the same parameters but have different implementations.

3.1.2 Benefits

The properties of OOT offer support for re-usability across multiple systems. Substantial re-work is required when objects or classes are re-used on other systems [16]. The scope of the problem for the design intent, the type of variation that the design must accommodate, and the anticipated changes that might occur have to be considered for the substantial re-work [16]. OOT analysis methods focus on the problem space and a single application by identifying the problem, objects required, messages required to be sent to objects and creating a sequence of messages as a solution to the problem. OOT offers organizational benefits which include [16]:

- mapping the problem space to the solutions through design patterns.
- reducing the proliferation of classes to enable the development of new systems.
- an extended support for systematic reuse.

3.2 SystemVerilog

3.2.1 Overview

SystemVerilog is an extension of the 1364 Verilog 2001 standards. It combines hardware description language features with features of OOT and verification. A combination of programming languages C and C++ along with hardware description languages VHDL and Verilog. It was originally introduced to be an extension to Verilog but later developed into a strong language for design, verification test benches, assertions, coverage, **D**irect **P**rogramming **I**nterface (**DPI**) and **A**pplication **P**rogramming **I**nterface (**API**). SystemVerilog is productive, readable and reusable. Its powerful features of constrained-random testing, assertion and checker monitors, and coverage-driven testing provide a complete verification environment. [17][15][18]

3.2.2 Benefits

The advantages of using SystemVerilog over Verilog or VHDL hardware description languages include[17][14][19][8][18]:

- Standardized and approved by Accellera and **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers (**IEEE**), provides wide and extended support to EDA tools and vendors.
- Increasing the productivity, readability, and re-usability of hardware descriptions.
- Providing a higher design and verification abstraction level.
- Providing a modular approach for integration thereby reducing costs and risks of adopting a new language.

- Eliminating the necessity of external verification tools.
- Extending the modeling of Verilog through direct programming interfaces by allowing C, C++, and SystemC through DPI.
- Reducing the overhead of **Verilog P**rogrammable **L**ogic **I**nterfaces (**Verilog PLI**)
- Employing directed, random and constrained-random stimulus testing.
- Monitoring messages and transactions though assertions, checkers and coverage-driven tests.
- Extending the Verilog data types and using interfaces to improve data encapsulation and abstraction.
- Enhancing process control by the use of fork, suspend and kill.
- Promoting cycle-based functionality by the use of clocking blocks and cycle-based attributes.
- Synchronizing inter-process communication through the use of semaphores and mailboxes.
- Promoting the use of dynamic re-sizable and associative arrays.

Chapter 4

Serial Peripheral and Wishbone Interface

4.1 Overview

Motorola developed the **S**erial **P**eripheral **I**nterface (**SPI**) in the year 1979 [20]. It is not a plug-and-play device and the interface type is configured for 3+N wires. This protocol is used for short distance communication and governed by the transmission of data streams. It is widely used in different industries due to its simplicity, low cost, and low power. It is a synchronous serial interface due to the presence of a clock [21]. It enables full duplex communication. It can only be configured as a single master and multiple slave protocol. The addressing mechanism is through the **S**lave **S**elect (**SS**) with no flow control and clock stretching. The transfer rate of this communication interface ranges from (n*1MHz to 10*n*1MHz). The communication of the SPI master and slave devices with the processor on a chip through the Wishbone bus is shown in Fig. 4.1.

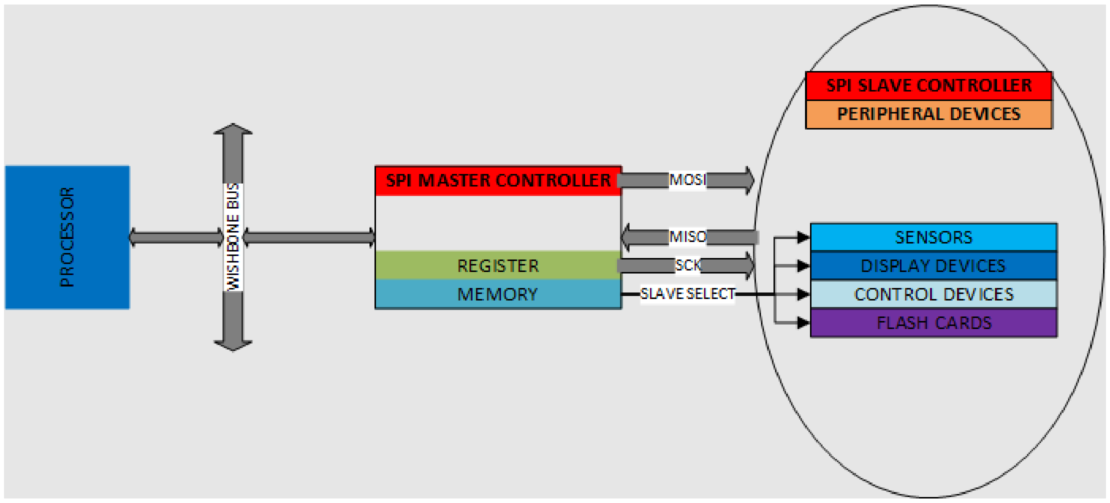


Figure 4.1: Processor-SPI Interaction

4.2 SPI Features

It is a widely used synchronous serial interface due to the following features which include [20][22][4]:

4.2.1 Data Transfer and Addressing

- It is a low to medium data transfer protocol with a variable transfer rate governed by the baud rate of the SPI Master device.
- It supports a user-defined word length and a frame data transfer enabling a full duplex communication between the master device and slave device.
- It supports a polling and an interrupt-driven mode transfer of data.
- It supports the automatic selection of the slave device through **A**uto **S**lave **S**elect (**ASS**).
- The addressing is through the SS pin of the SPI device.
- It supports synchronous serial communication through the presence of a clock signal.
- It may be used as a transmitter or receiver, with or without FIFO data transfer, and with or without digital filters.

- It is synchronous to positive edge clocking for scan-path insertion and does not have any internal 3-state elements.

4.2.2 Synchronization

- The data transfer and the shifting and sampling of data are synchronized on the **S**erial **C**loc**K** (**SCK**) with configurable clock polarity, clock phase, transmission and reception positive edge clocking.

- It supports a user-defined baud rate and clock rate.

4.2.3 Error Detection and Data Integrity

- It supports the detection of overflow errors during a data transfer are through internal flags. Under-run errors are monitored and detected through the output from the slave to the master.

- It supports the elimination of the input spikes which are shorter than a user-defined number of clock cycles.

4.2.4 Host-side interface

- It enables a simple handshaking mechanism and a full duplex transfer which can be adapted to any standard chip.

4.2.5 Performance

- It has a simple user-defined **F**irst-**I**n-**F**irst-**O**ut (**FIFO**) to improve the performance in the transfer of data.

- It supports lower power consumption by the disconnection of SPI Master and Slave devices from the system through a power conservative state.

4.3 SPI Advantages, Disadvantages and Applications

4.3.1 Advantages

SPI is advantageous over other serial communication protocols as[4]:

- It provides good signal integrity and speed due to push-pull drivers.
- It is flexible for the bits transferred.
- It is not limited to transfer of 8-bit data words alone.
- It supports full duplex data transfer and higher throughput.

4.3.2 Disadvantages

SPI is disadvantageous over other serial communication protocols as[4]:

- It does not have a flow control mechanism by the slave device.
- It does not receive any acknowledgment from the slave device.
- There are no in-band addressing chip select signals and the addressing is through SS only.
- It has more interfaces when compared to other serial communication protocols.
- It does not support multiple master devices.
- It does not enable parallel communication and data transfer is only one-bit at a time.

4.3.3 Applications

It is widely used in a number of applications in different industries and domains. Some of the applications of the SPI include sensors, control devices, communication systems, Memory and SD cards.

Table 4.1: Applications[4]

Sensors	Control Devices	Communications	Memory	Clocks	Displays
Temperature	CODECs	Ethernet	Flash	Real Time Clocks	LCD
Pressure	DACs	USB	EEPROM		
Touch Screens	Digital Potentiometers	CAN	SD Card		
		USA RT	MMC Card		
Controllers		IEEE 802.11			
ADCs		IEEE 802.15.4			

4.4 SPI Interfaces

A standard SPI is designed with four interfaces which include an optional slave select, Master-In-Slave-Out, Master-Out-Slave-In, Slave Select, and Serial Clock. Different designs have different naming conventions for these signals and the standard naming convention is followed in this research [4]. The interface signals of the SPI is showcased in Fig. 4.2.

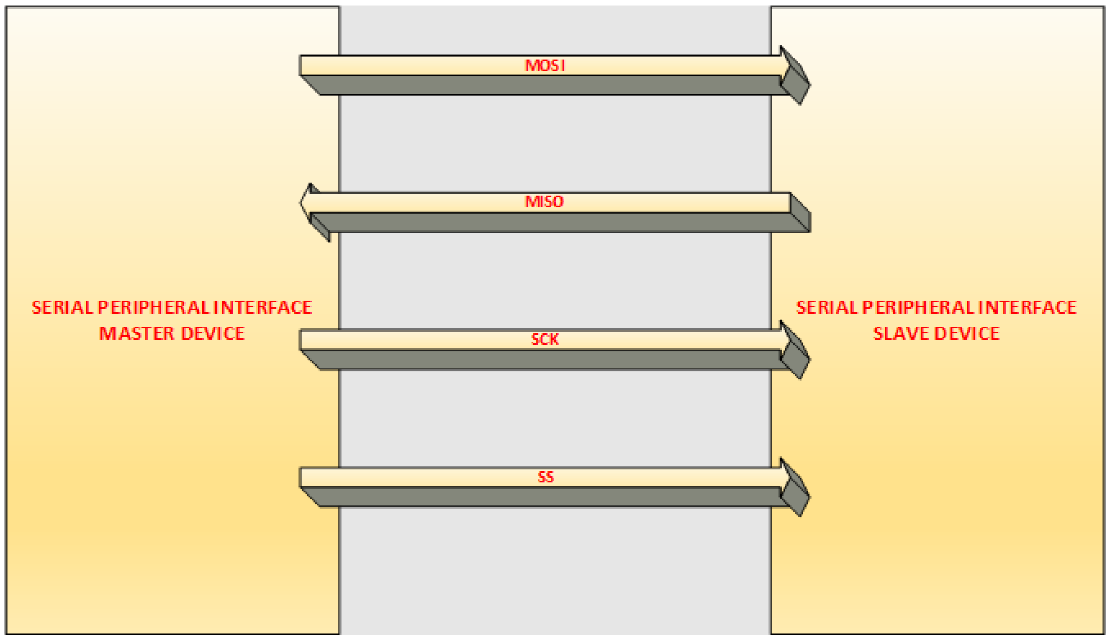

MOSI
MISO
SERIAL PERIPHERAL INTERFACE MASTER DEVICE
SCK
SERIAL PERIPHERAL INTERFACE SLAVE DEVICE
SS

Figure 4.2: SPI Interfaces in a single master-single slave device configuration

4.4.1 Master Out Slave In (MOSI)

This signal is transmitted as an output from the master to the slave as an input. It is used to transfer the data in one direction. In a typical scenario, the most significant bit is sent first on this data line.

4.4.2 Master In Slave Out (MISO)

This signal is received as an input to the master device from the slave device. It is used to transfer the data in the other direction. In a typical scenario, the most significant bit is sent first in this data line. A high impedance state is observed for this signal if there is no slave selected by the master.

4.4.3 Slave Select (SS)

This signal selects the slave device by switching to the active low state. A data transfer is initiated only after the slave is selected and must remain in this state till the data transfer is complete.

4.4.4 Serial Clock (SCK)

This signal synchronizes the data samples and the data shifts on MOSI and MISO to the positive and negative edges of the generated clock. This signal can only be generated by the master device and is an input to the slave. A byte of data or information can be transferred between master and slave devices during a sequence of 8 clock cycles.

4.5 SPI Data Transfer Modes and Formats

A two-way communication is established between the microcontroller or microprocessor and the peripheral devices. The data word or character is simultaneously shifted out from the master to slave and shifted in from the slave to master serially one bit a clock cycle. The data sampling and shifting are synchronized by the serial clock and only after a slave peripheral device is selected by the microcontroller or microprocessor [23]. Two bits in the SPI control register control the clock phase and polarity. There are four combinations of the clock phase and polarity as shown in Tab. 4.2 define the modes of operation of the data transfer. The phase and polarity should be identical to both the master and slave through out the transfer. As SPI is synchronous to the serial clock, the data shift occurs on one edge of the serial clock (either positive or negative) and the data capture is on the other edge. Ideally, there are only two data transfer formats based on the clock phase.

Table 4.2: Modes of Operation

Mode	Polarity (CPOL)	Phase (CPHA)	Data Shift	Data Sample
0	0	0	Falling (Negedge)	Rising (Posedge)
1	0	1	Rising (Posedge)	Falling (Negedge)
2	1	0	Rising (Posedge)	Falling (Negedge)
3	1	1	Falling (Negedge)	Rising (Posedge)

4.5.1 CPHA Equals Zero Transfer Format

Fig. 4.3 depicts a sample timing diagram when the phase is zero. As the SCK, MOSI and MISO pins are inter-connected between master and slaves, it may interpret either a master or a slave timing diagram. An 8-bit data transfer requires 8 clock cycles as one bit is transferred at a time. This is denoted by the SCK Cycle. When the polarity is one, the data shift happens on the positive edge (posedge) of the serial clock and the data capture happens on the negative edge (negedge) of the serial clock. When the polarity is zero, the data shift happens on the negedge of the serial clock and the data capture happens on the posedge of the serial clock. SS has to be triggered to active low before the data shift and capture starts as it ensures the selection of the slave device for communication. The SS must be de-asserted and re-asserted to to avoid a write collision error between successive serial byte transfers, when the clock phase is zero[23]. Bit 7 represents the **M**ost **S**ignificant **B**it (**MSB**) and bit 0 represents the **L**east **S**ignificant **B**it (**LSB**). A general tendency of data transfer is to send the MSB first on the MOSI or MISO data lines.

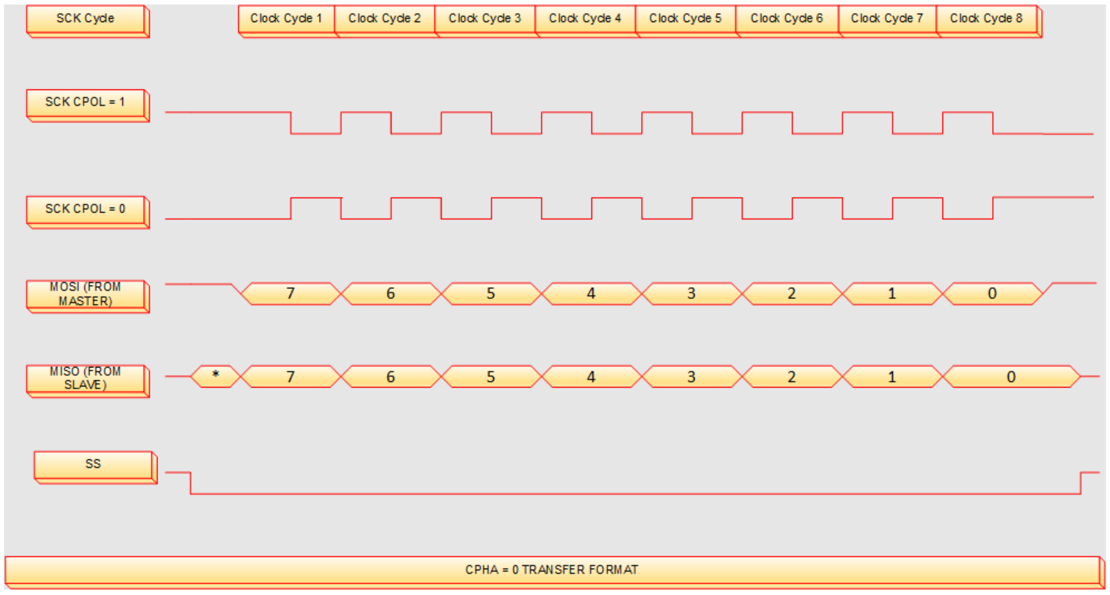


Figure 4.3: CPHA = 0 Timing Diagram

4.5.2 CPHA Equals One Transfer Format

Fig. 4.4 depicts a sample timing diagram when the clock phase is set to one. As the SCK, MOSI and MISO pins are inter-connected between the master and slaves, it may interpret either a master or a slave timing diagram. An 8-bit data transfer requires 8 clock cycles as one bit is transferred at a time. This is denoted by the SCK Cycle. When the polarity is one, the data shift happens on the negedge of the serial clock and data capture happens on the posedge of the serial clock. When the polarity is zero, the data shift happens on the posedge of the serial clock and the data capture happens on the negedge of the serial clock. SS has to be triggered to active low before the data shift and capture starts as it ensures the selection of the slave device for communication. The SS can remain in the active low state at all times, when the clock phase is 1 [23]. Bit 7 represents the MSB and bit 0 represents the LSB. A general tendency of data transfer is to send the MSB first on the MOSI or MISO data lines.

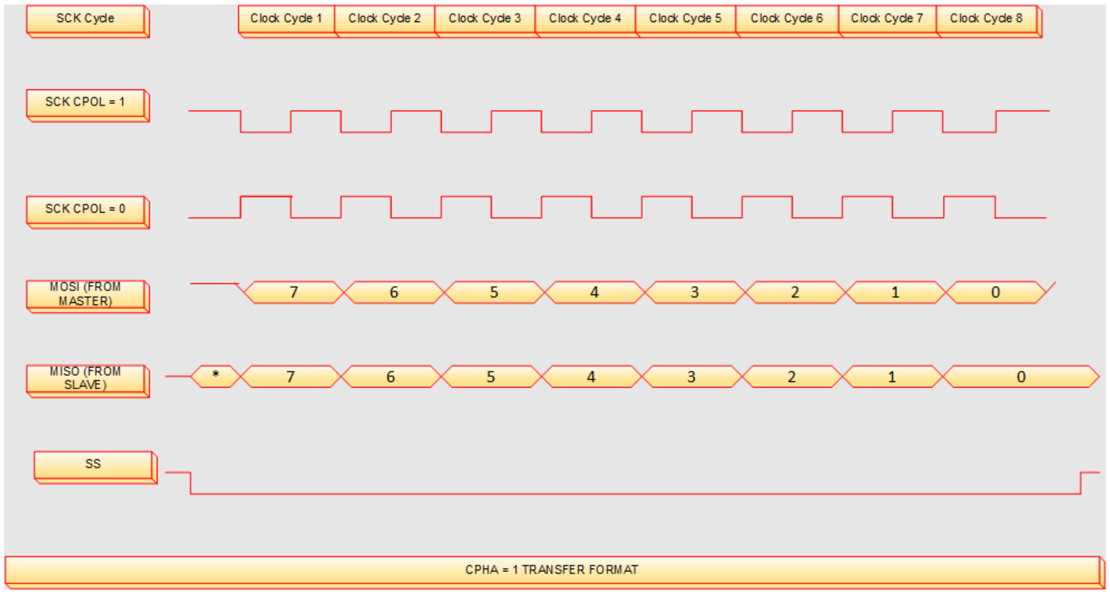


Figure 4.4: CPHA = 1 Timing Diagram

4.6 SPI Sub-System

Fig. 4.5 showcases the internal communication logic between the master and slave device registers. During an 8-bit transfer in the SPI, an 8-bit character is shifted through one pin from the master device and another 8-bit character is shifted in through the second data pin from the slave device simultaneously. It can be understood as an 8-bit shift register existing in the master and the slave respectively. These two shift registers are connected as a 16-bit shift register with a circular operation to ensure the data exchange. In Fig. 4.5, a divider is used to generate the clock frequency of the SPI from the Wishbone clock as Wishbone and SPI operate at different clock frequencies. The signals and their bit positions in the different registers are described in the Sec. 4.7.

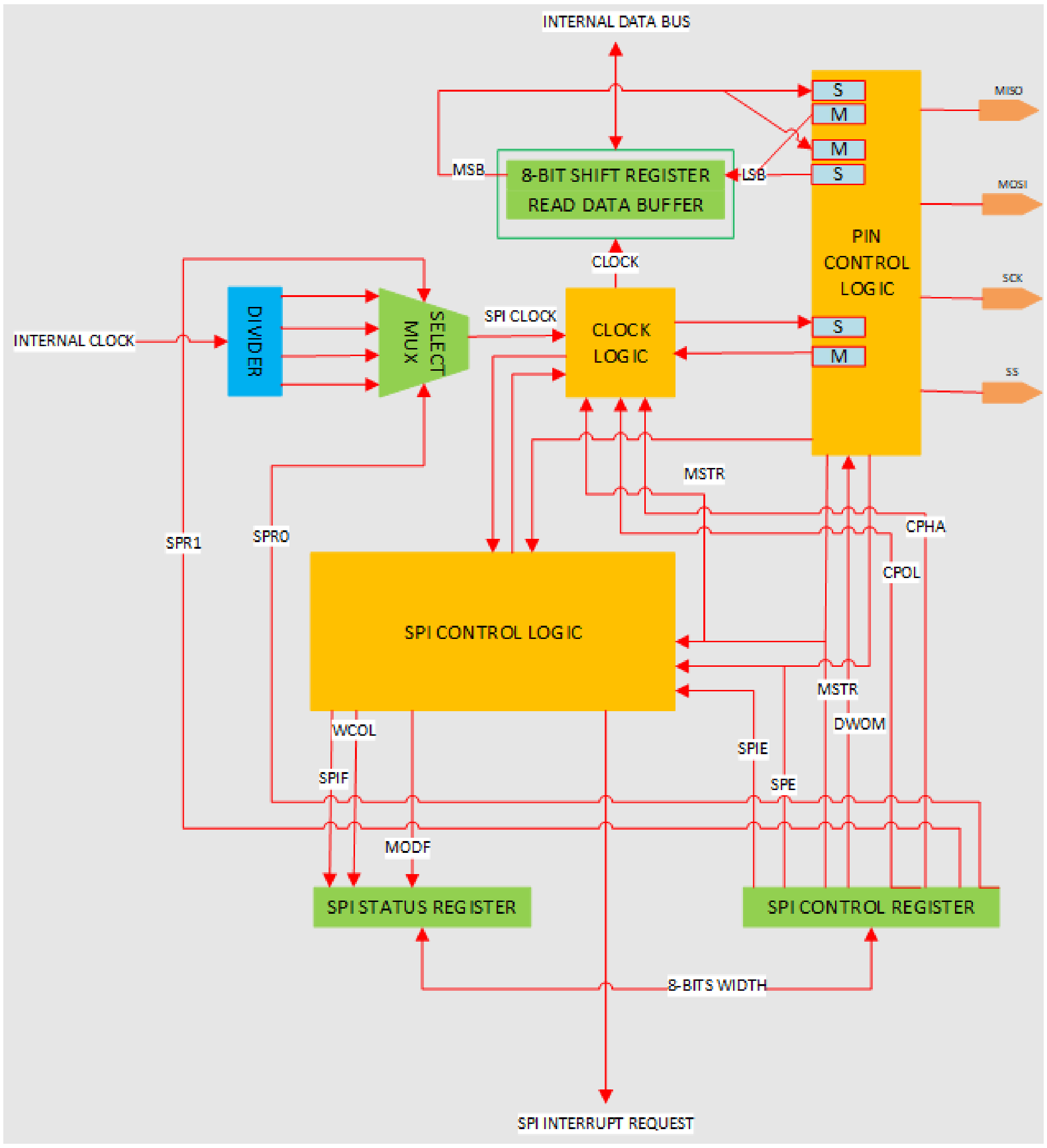


Figure 4.5: SPI Sub-System

The 8-bit shift register and the read data buffer is the central block of the Fig. 4.5. It is single buffered in the transmit direction to ensure a new character is transferred only after the transfer of the previous character is complete and is double-buffered in the receive direction [23]. The SPI status and control registers of the sub-system as shown in Fig. 4.5 are used to monitor and control the status of the transaction of the character. The transmit and receive of data are synchronized to the serial clock. The pin control logic monitors the SPI interface signals. 'S' represents 'Slave' and 'M' represents 'Master' in the Pin Control Logic. Fig. 4.6 shows the bit serial operation of an 8-bit SPI master and an 8-bit SPI slave communication. It also shows as to how it can be linked to the working of a 16-bit circular shift register.

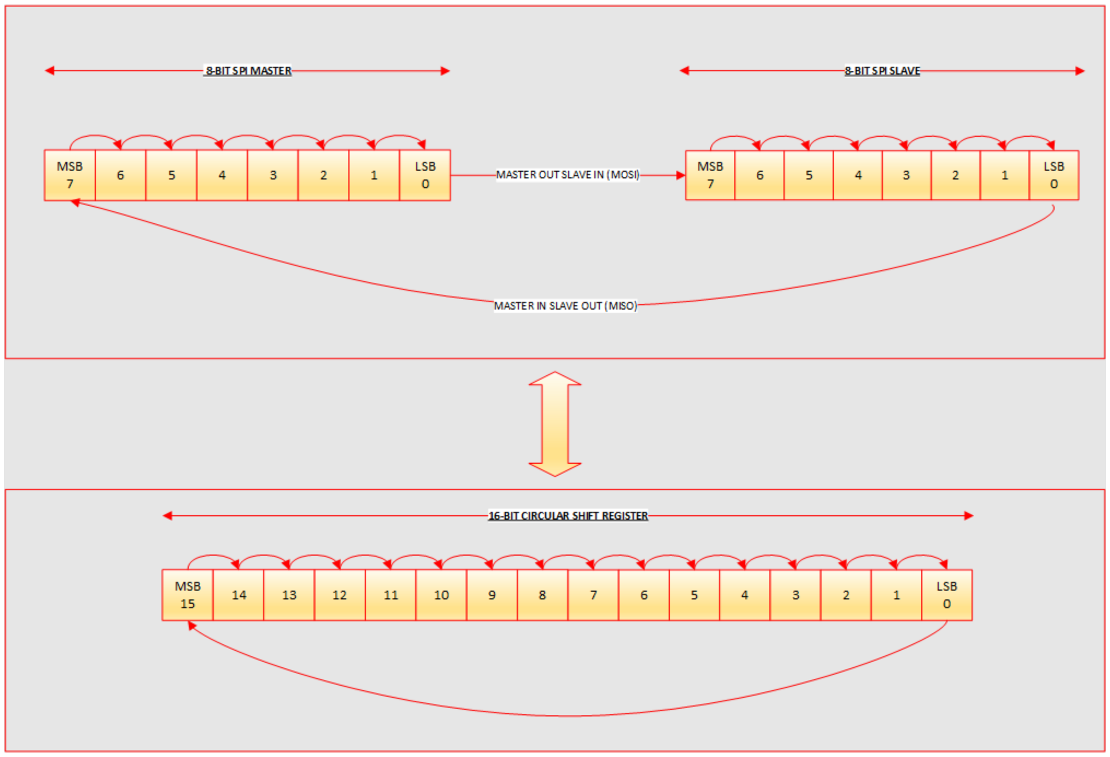

8-BIT SPI MASTER
8-BIT SPI SLAVE
MSB 7
6
5
4
3
2
1
LSB 0
MASTER OUT SLAVE IN (MOSI)
MASTER IN SLAVE OUT (MISO)
16-BIT CIRCULAR SHIFT REGISTER
MSB 15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
LSB 0

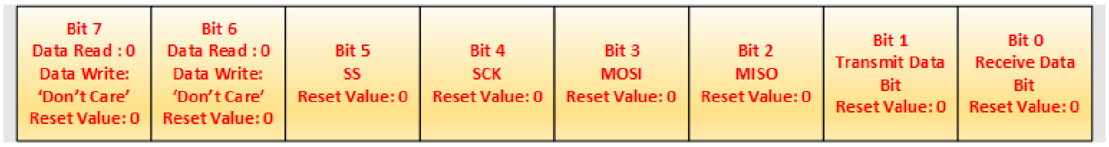
Figure 4.6: Data Transmission

4.7 SPI Registers

Three registers namely the data direction control register (data shift register), the SPI control register and the SPI status register are used to control the data transactions in the SPI communication interface.

4.7.1 Data Direction Control Register

This register reads or writes the data at any given instance of time. When the **SPI E**nable pin (**SPE**) is high, bits 5 to 2 are used by the SPI sub-system for the pins [24]. Bit position 1 and bit position 0 are used for transmit and receive operations. A pictorial representation of the bits in an 8-bit SPI register is shown in the Fig. 4.7 register. The positions of the signal representations may change as the data width increases.



Bit 7 Data Read : 0 Data Write: 'Don't Care' Reset Value: 0	Bit 6 Data Read : 0 Data Write: 'Don't Care' Reset Value: 0	Bit 5 SS Reset Value: 0	Bit 4 SCK Reset Value: 0	Bit 3 MOSI Reset Value: 0	Bit 2 MISO Reset Value: 0	Bit 1 Transmit Data Bit Reset Value: 0	Bit 0 Receive Data Bit Reset Value: 0

Figure 4.7: Data Direction Control Register

- When SPE is high and **MaSTeR (MSTR)** is low, bit position 5 is the SS pin regardless of its value. When SPE and MSTR are both high, the function of SS depends on the value in bit position 5 of the register. Value '0' indicates the use of SS to detect mode fault errors and value '1' indicates the use of SS as a general purpose output and this output will not be affected by the SPI sub-system [23].

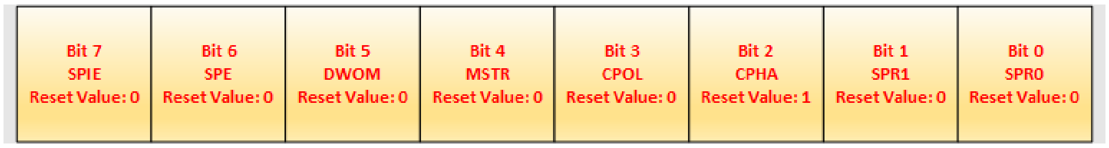
- Bit position 4 of the register indicates the SCK. This bit is set high when SPI is used as a master. This bit represents SCK regardless of its value when SPI is used as a slave [23].

- Bit position 3 of the register indicates the MOSI. It is switched to a high state if operated as a master and the master initiates a transaction. The value is of no importance if operated as a slave [23].

- Bit position 2 of the register indicates the MISO. It is switched to a high state if operated as a slave. This value is of no importance if operated as a master [23].

4.7.2 Control Register

This register is the brain of the SPI sub-system used for configuration of SPI. This register is read from or written to at any given instance of time [24]. A pictorial representation of the bits in an 8-bit SPI register is shown in Fig. 4.8. The positions of the signal representations may change as the data width increases.



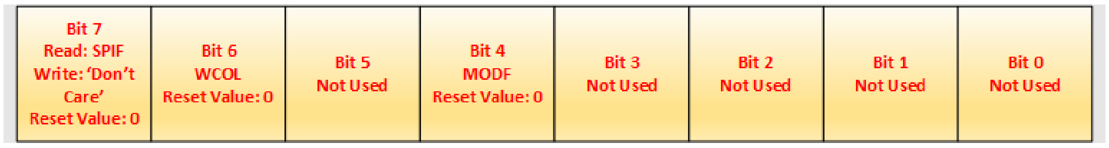
Bit 7 SPIE Reset Value: 0	Bit 6 SPE Reset Value: 0	Bit 5 DWOM Reset Value: 0	Bit 4 MSTR Reset Value: 0	Bit 3 CPOL Reset Value: 0	Bit 2 CPHA Reset Value: 1	Bit 1 SPR1 Reset Value: 0	Bit 0 SPR0 Reset Value: 0

Figure 4.8: Control Register

- Bit position 7 indicates the enabling of SPI interrupts. It is switched to a low state to disable the interrupts and use polling to sense the flags in the status register. It is switched to a high state to enable the interrupts [23].
- Bit position 6 indicates the enabling of the SPI sub-system. It is enabled when set high.
- Bit position 5 represents the Wired OR-Mode select. This bit is set low to output push-pull drivers and set high to output open drain drivers [23].
- Bit position 4 is used to indicate the operation of SPI as either a master or a slave device. It is set high to enable SPI master operation[25][26].
- Bit position 3 is used to indicate the serial clock polarity. It is set high to enable the selection of active low clocks and vice versa[25][26].
- Bit position 2 indicates the serial clock phase to select the data transfer format[25][26].
- Bit positions 1 and 0 indicate the bit rate select bits to select the clock output from the divider in the SPI sub-system.

4.7.3 Status Register

This register can only be read. Write access to this register is unavailable and it contains the status flags indicating the completion of data transfer and transfer overflow and under-run system errors. The bits in this register are set automatically based on the events of the SPI [24]. A pictorial representation of the bits in an 8-bit SPI register is shown in Fig. 4.9. The positions of the signal representations may change as the data width increases.



Bit 7 Read: SPIF Write: 'Don't Care' Reset Value: 0	Bit 6 WCOL Reset Value: 0	Bit 5 Not Used	Bit 4 MODF Reset Value: 0	Bit 3 Not Used	Bit 2 Not Used	Bit 1 Not Used	Bit 0 Not Used

Figure 4.9: Status Register

The SPI sub-system as shown in Fig. 4.5 is only concerned with the bits 7, 6 and 4. All the other bits are not used and always set low.

- Bit position 7 represents the flag for the completion of data transfer. It reaches a high state automatically at the end of the transfer. It is automatically cleaned before a new data transfer by reading from the register with this bit in high state.
- Bit position 6 represents the flag for a write collision error. It reaches a high state automatically if the data direction register is written while a transfer is in progress. It is automatically cleaned before a new data transfer by reading from the register with this bit in high state.
- Bit position 4 indicates the mode fault error and reaches a high state the device is operated as a master and SS goes active low. It is automatically cleaned by reading from this register with this bit in the high state followed by a write to the control register.

4.8 Beginning and Ending SPI Transfers

- The clock phase data transfer format and configuration of the SPI as a master or a slave device determines the beginning and the ending period of the transfer.
- An initiation delay occurs at the beginning and end of every SPI data transfer.
- The initiation delay is affected by the clock phase transfer format and the SPI clock rate.
- Clock phase does not affect the initiation delay but has an effect on the initial state of the serial clock
- The SPIF bit flag in the SPI status register signifies the end of the transfer. This flag is set after the transfer of data based on the SPI clock rate.

4.9 Wishbone Bus Interface

Wishbone is a flexible SoC interconnection architecture used with different IP cores. It enables a faster design re-usability by avoiding integration issues on a SoC. Wishbone interface is used as it is independent of the various logic signaling levels of different IP cores on a chip. It employs a simple Master-Slave architecture like the SPI which makes the design interface easier. Wishbone also employs parallel communication between devices which results in faster execution. All the Wishbone Master and Slave devices use an **INTERCON**nection interface (INTERCON). INTERCON, as shown in Fig. 4.10 contains standard logic circuits which can be used by the masters and slaves[27] [28][29].

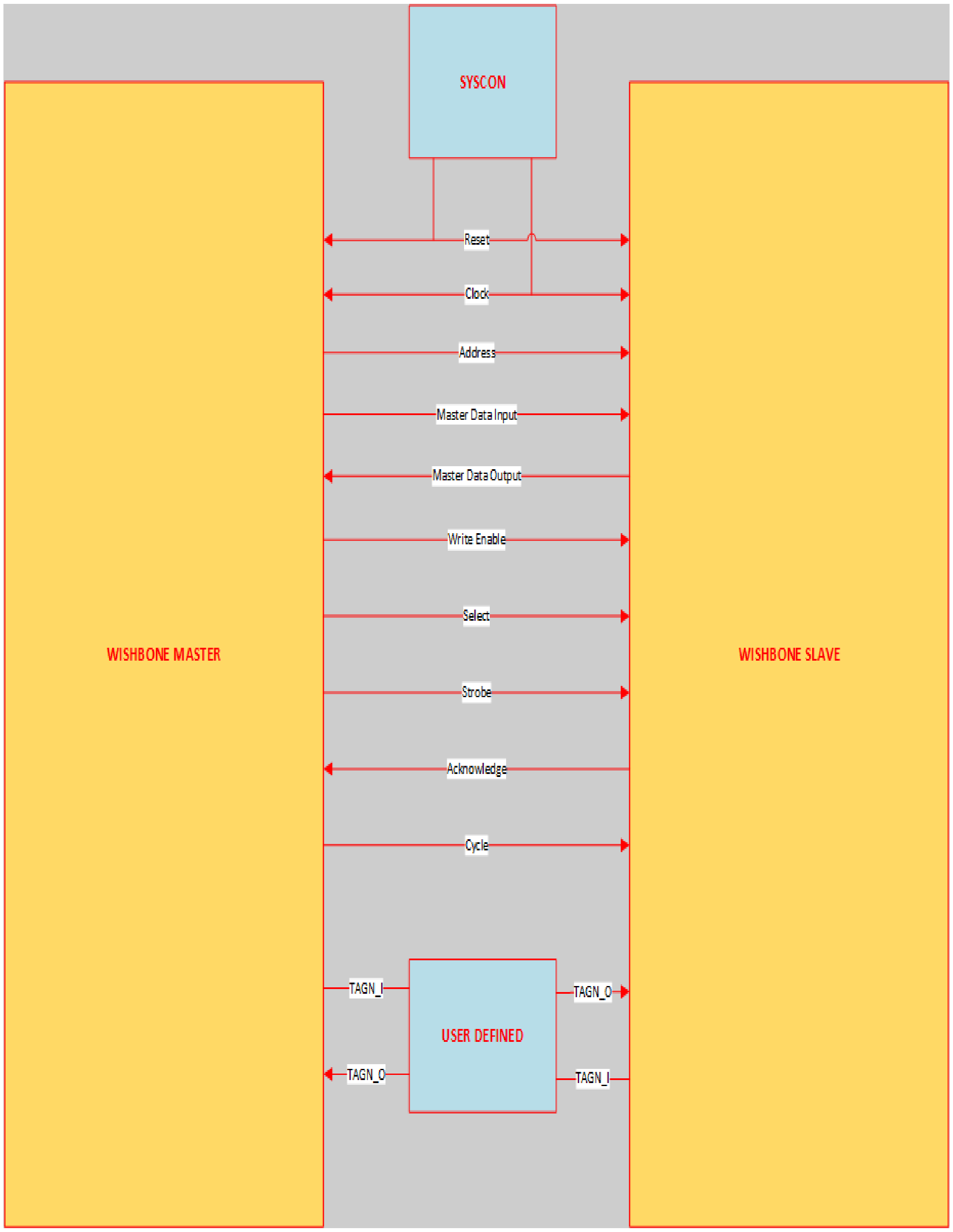


Figure 4.10: Wishbone INTERCON[1]

Wishbone Master is considered as the host to the SPI. It implies that SPI master device is a wishbone compliant slave device. The Wishbone signals used in interfacing with the SPI are tabulated in Tab. 4.3.

Table 4.3: Wishbone Signals[1, 5]

Signal	Width	Direction	Description
wb_clk_i	1	Input	System Clock
wb_rst_i	1	Input	Active high synchronous Reset
wb_adr_i	32	Input	Address
wb_dat_i	128	Input	Data to Core
wb_dat_o	128	Output	Data from Core
wb_sel_i	16	Input	Byte Select
wb_we_i	1	Input	Write Enable
wb_stb_i	1	Input	Strobe
wb_cyc_i	1	Input	Bus cycle
wb_ack_o	1	Output	Bus cycle acknowledge
wb_err_o	1	Output	Bus cycle error
wb_int_o	1	Output	Interrupt

The wishbone logic is synchronous to the system clock. Master initiates the bus transaction. Setting the wb_rst_i to active low restarts the core, presets internal registers to the default values and sets state machines to the initial or default known state. The host controller assists the core if the interrupt wb_int_o is asserted. When asserted, a valid bus cycle is checked using the signal wb_cyc_i and a valid transfer cycle is checked using the logical AND of the signals strobe wb_stb_i and write-enable wb_we_i. The core responds to the host controller only if there is a valid bus transfer cycle. A valid binary coded address is passed to the core using the signal wb_adr_i which signifies the start of the bus transaction. The assertion of the signal wb_we_i signifies a Wishbone write cycle else a Wishbone read cycle. Data is sent from the host controller to the core through the signal wb_dat_i and the core responds to the host controller with the data through the signal wb_dat_o. A valid bus cycle is terminated normally by asserting the acknowledged output wb_ack_o. The signal wb_err_o is used to signify an error in the bus transaction cycle. [1][27][30]

Wishbone is used as it enables parallel communication in the chip. The parallel data in wishbone is converted into serial data for communication between the SPI master and slave devices. The serial data from the slave device is converted back into parallel data before it is sent to the Wishbone bus.

4.10 System Architecture

Fig.4.11 shows the top level system architecture. The communication interface consists of a wishbone interface and the SPI sub-system. The SPI sub-system includes a clock divider and a shift register. The clock divider is used as the SPI and Wishbone operate at different clock frequencies. The shift register function is used for the data word or character exchange between devices. Wishbone bus communicates with the processor and the communication is parallel. The data word from the processor is parallel and is converted into serial data for the data exchange between devices. The serial data is converted back into parallel data to be sent to the processor.

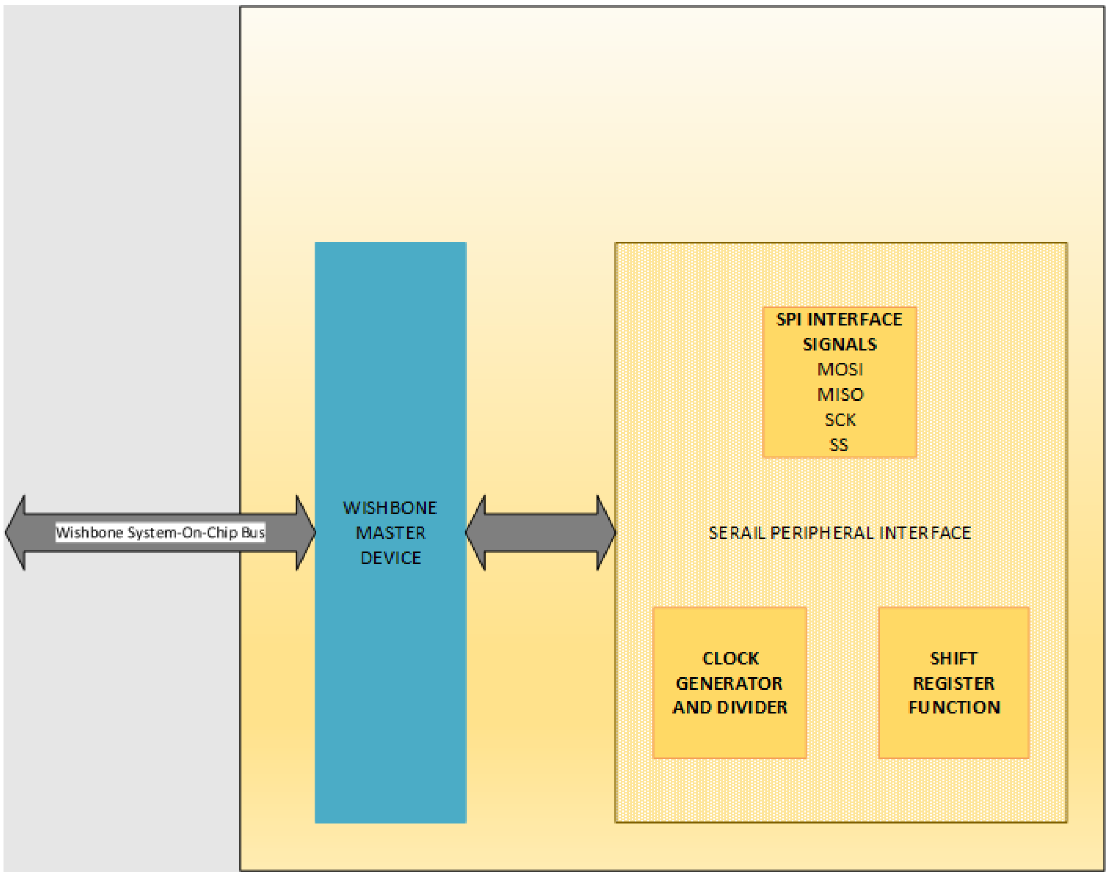

Wishbone System-On-Chip Bus
WISHBONE MASTER DEVICE
SPI INTERFACE SIGNALS
MOSI
MISO
SCK
SS
SERAIL PERIPHERAL INTERFACE
CLOCK GENERATOR AND DIVIDER
SHIFT REGISTER FUNCTION

Figure 4.11: Top-Level Architecture[2]

Fig.4.12 shows an extended architecture of the system with the interaction of the SPI registers with the wishbone interface. As discussed in Sec. 4.7, the SPI registers include the control register, the status register, and the data direction control register.

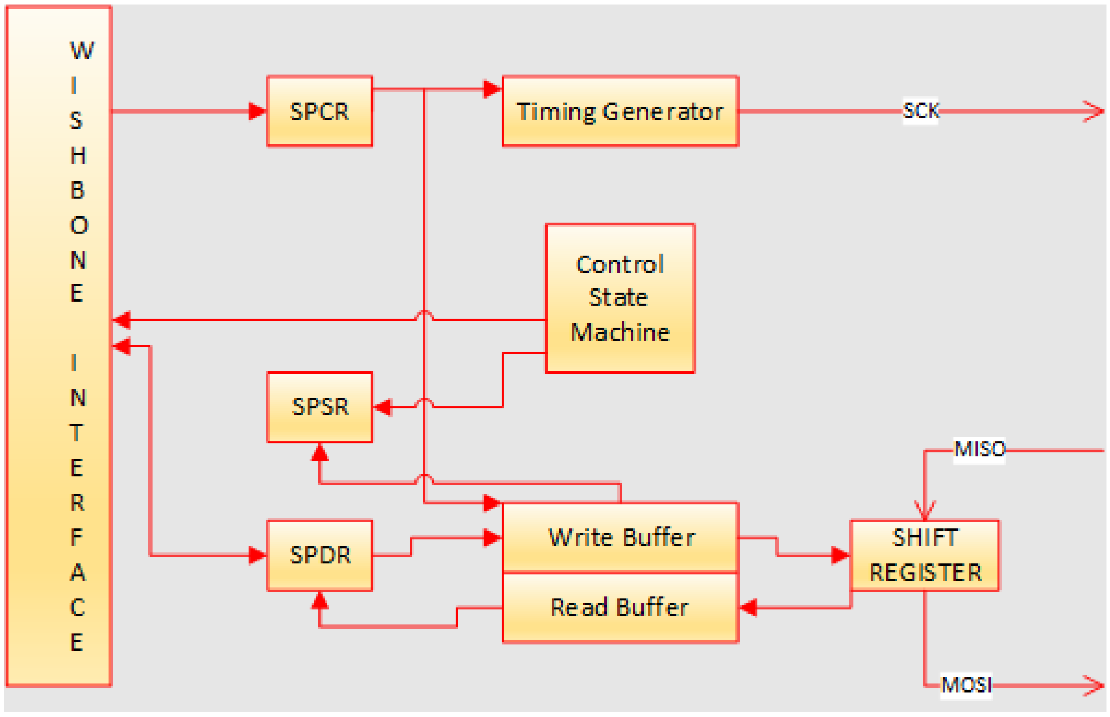


Figure 4.12: System Detailed Architectural Block Diagram[3]

Chapter 5

Testbench Methodology and Results

5.1 Testbench Methodology

A testbench is an additional entity that provides a repeatable set of stimuli to the design under test to verify its functional correctness. It is portable across different simulators and checks if the design meets the system and requirement specifications. It may be a simple hardware description file with clock and input constraints or a complex environment for error checks, conditional checks, input constraints, and output constraints. It provides an early indication of the operation and performance of components of the design. A flat testbench is a simple Verilog or VHDL file with clock and input constraints. A flat testbench generates inputs, resets and configures the design under test, runs tests, capture outputs, verifies correctness and reports errors. A SystemVerilog testbench is a complex modular environment with a higher level abstraction of the design, explicit design intent, and concise expressions. A SystemVerilog testbench focuses on the expected behavior of the system, as the input test combinations are automatically generated by the test generators.

A layered testbench approach using SystemVerilog was followed in the development of the environment due to the benefits mentioned in Sec. 2.3 and Sec. 3.2 of this research paper. The developed testbench is divided into signal, command, functional and scenario layers using object-oriented technology concepts of classes, objects, and use-cases. The testbench components which were developed for the verification environment using the bottom-up methodology are described in the sub-sections of this chapter. Fig. 5.1 shows the testbench environment. Fig. 5.2 shows the testbench environment with interface signals.

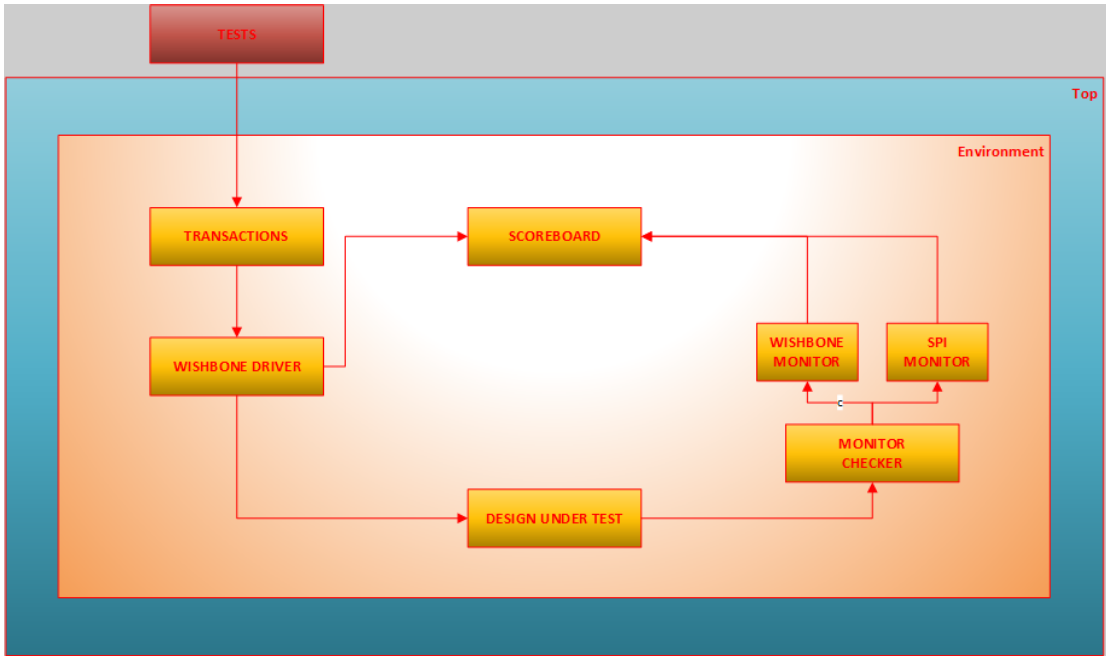


Figure 5.1: Testbench Environment

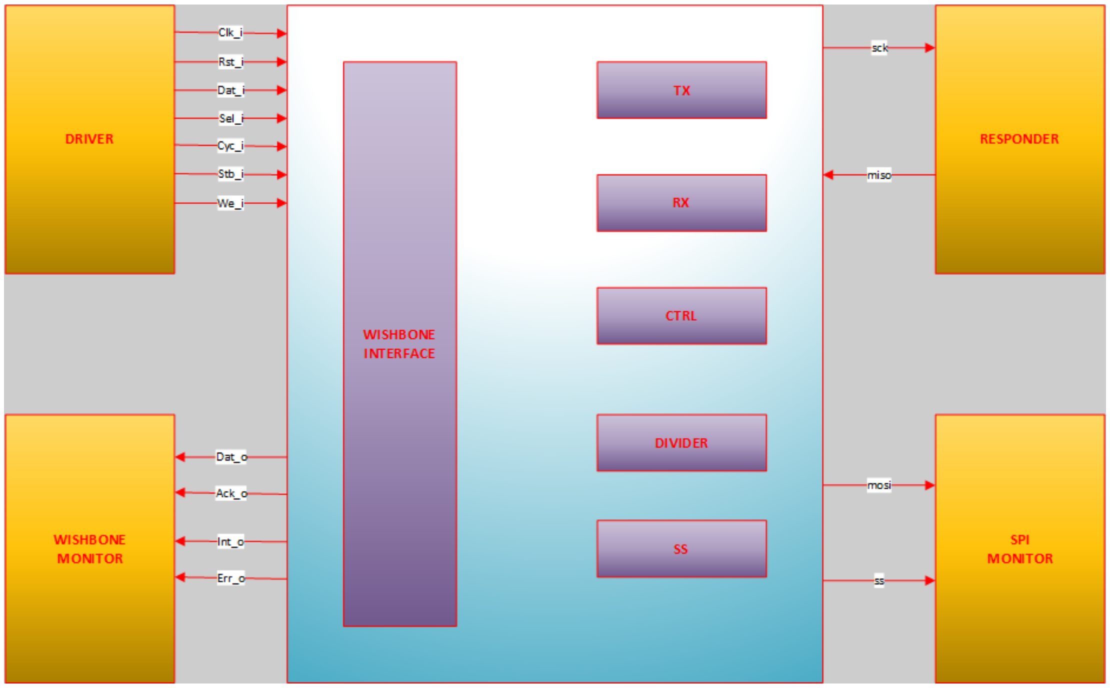


Figure 5.2: Testbench Environment with Interface Signals

5.2 Testbench Components

5.2.1 Interface

SystemVerilog provides a useful construct named interface which helps the testbench environment to interact with the design under test. The interface block will have all the Wishbone and SPI signals along with modports for each protocol.

5.2.2 Transaction

The interaction between the testbench components happens through transaction objects. The transaction objects include the address of the data or transaction, the payload and the direction of the data (either transmission or reception). The constrained-random transactions are passed on to the driver via transaction channels. Separate classes are created for the transactions to monitor and check Wishbone and SPI signals. The SPI design is capable of transferring up to 512 bits. As it isn't possible to determine the number of bits to be transferred on the SPI interface, Wishbone monitor will store each 128-bit data to the SPI registers. The 512-bit data from the SPI registers along with the character length from the control register is sent to the scoreboard via the driver to check the SPI outputs.The transaction objects created are randomized using the randomize function of SystemVerilog. This is used to automate the process of generating all possible input stimulus and constrained-random stimulus. This approach saves time and helps in focusing on the testbench framework and behavior of the system.The stimulus is passed to the driver through mailboxes.

5.2.3 Driver

A driver is used to convert the transactions into read and write operations to be sent to the core. In this environment, it is converted into Wishbone read and write cycle operations as Wishbone is the host controller of the system. The signals are put to a known state through the Wishbone reset task. The driver interacts with the design under test through the interface by creating it virtually.

5.2.4 Monitor

The monitor monitors transactions at different events and also includes a checker. It consists of two tasks which fired in parallel as soon as the transfer of data is initiated. One task monitors and checks the wishbone outputs and the other task monitors and checks the SPI interfaces. In order to make it simpler, the monitors are created as tasks in the same class.

5.2.4.1 Wishbone Monitor

A task is created to monitor the transactions on the host-side. The purpose of a wishbone monitor is to send the wishbone transactions to the scoreboard for analysis checks. The structure of the wishbone monitor is similar to that of the driver except for the transactions to the scoreboard. This monitor is used to only check the transactions on the host side.

5.2.4.2 SPI Monitor

A task is created to monitor the transactions on the core-side. The purpose of the SPI monitor is to send the SPI transactions to the scoreboard for analysis checks. The SPI monitor generates a new transaction to be sent, waits for the transfer to be finished, makes sure the transfer is complete, and ends the transmission. This is used to monitor the SPI signals.

5.2.5 Scoreboard

The scoreboard is used to compare the actual data with the expected data. This is done by creating a mask based on the number of bits transmitted and compare the data. The scoreboard receives the transactions through mailboxes and the character length from the SPI control registers and it performs a compare data check. .

5.2.6 Environment

All the components developed are brought together in a unified testbench environment called the environment. These components are created in the environment in the order of their instantiation virtually through the interface as it binds the testbench and the design. A reference is created to the SPI design as it requires specific initialization before running a test which is done using the reset functionality.

5.2.7 Test Case

The test case instantiates the environment and is of the file type program. A *program* block instantiates the design wrapper and the top-level class-based environment. It cannot have the instantiation of modules, interfaces, or other programs[14]. SystemVerilog considers this as a test module to run the different scenarios and the simulation terminates when every block in the *program* is completed. All the tasks from the respective classes are called to enable a successful operation at the top-level of the system.

5.2.8 Top-Level

A top-level module is created to generate the clock, instantiate the interface, design under test and the test case. It ensures the components are called the right way.

5.3 Testbench Results

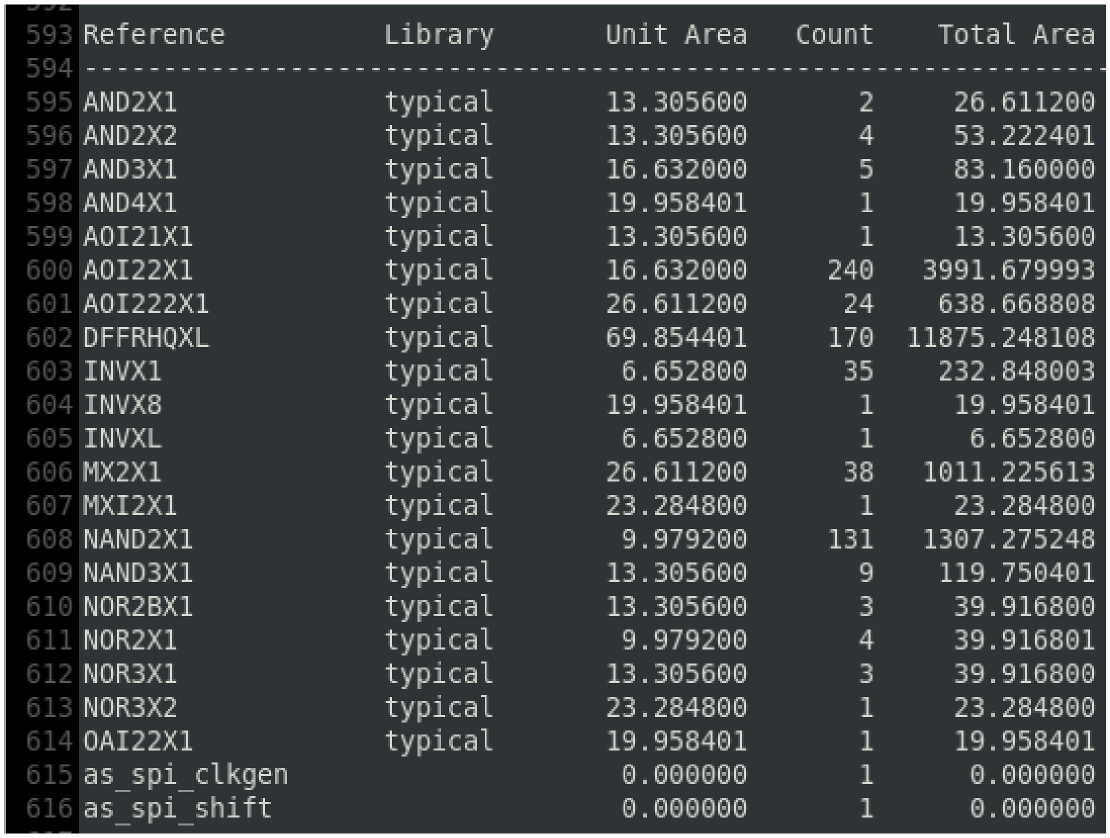
5.3.1 Design's Area, Power and Module Hierarchy

As this project is focussed on creating a functional verification environment, an open cores existing design was updated to the latest specifications in the TSMC 0.18 micrometer technology and only the total number of cells, total cell area and total dynamic power of the final design were captured in Tab. 5.1.

Table 5.1: Area and Power Details

Type	Value
Number of Cells	678
Number of Combinational cells	505
Number of Sequential Cells	173
Total Cell Area	19585.84
Total Dynamic Power	6.676 mW
Cell Internal Power	3.913 mW
Net Switching Power	2.763 mW
Cell Leakage Power	76.3688 mW

The module design hierarchy is showcased in the Fig. 5.3.



```
Reference          Library       Unit Area    Count     Total Area
-------------------------------------------------------------------
AND2X1             typical       13.305600        2      26.611200
AND2X2             typical       13.305600        4      53.222401
AND3X1             typical       16.632000        5      83.160000
AND4X1             typical       19.958401        1      19.958401
AOI21X1            typical       13.305600        1      13.305600
AOI22X1            typical       16.632000      240    3991.679993
AOI222X1           typical       26.611200       24     638.668808
DFFRHQXL           typical       69.854401      170   11875.248108
INVX1              typical        6.652800       35     232.848003
INVX8              typical       19.958401        1      19.958401
INVXL              typical        6.652800        1       6.652800
MX2X1              typical       26.611200       38    1011.225613
MXI2X1             typical       23.284800        1      23.284800
NAND2X1            typical        9.979200      131    1307.275248
NAND3X1            typical       13.305600        9     119.750401
NOR2BX1            typical       13.305600        3      39.916800
NOR2X1             typical        9.979200        4      39.916801
NOR3X1             typical       13.305600        3      39.916800
NOR3X2             typical       23.284800        1      23.284800
OAI22X1            typical       19.958401        1      19.958401
as_spi_clkgen                     0.000000        1       0.000000
as_spi_shift                      0.000000        1       0.000000
```

Figure 5.3: Module Design Hierarchy

5.3.2 Code Methodology

A correct flow had to be followed to ensure the correct working of the verification environment of the wishbone-compliant SPI master and slave. Classes and objects were created for each testbench component described in Sec. 5.2. As it can be seen in the Fig. 5.4, the top-level module instantiates the interface and test case classes along with the design under test. An object of the environment was created in the class 'testcase' which instantiates the environment and the objects inside the environment. The created environment class instantiates the driver though calls to the tasks reset and drive, the monitor through the main task, the scoreboard through the main task. The constrained-random transactions are sent to the driver, monitor and scoreboard classes. The simulation ends with $finish in the top-level class.

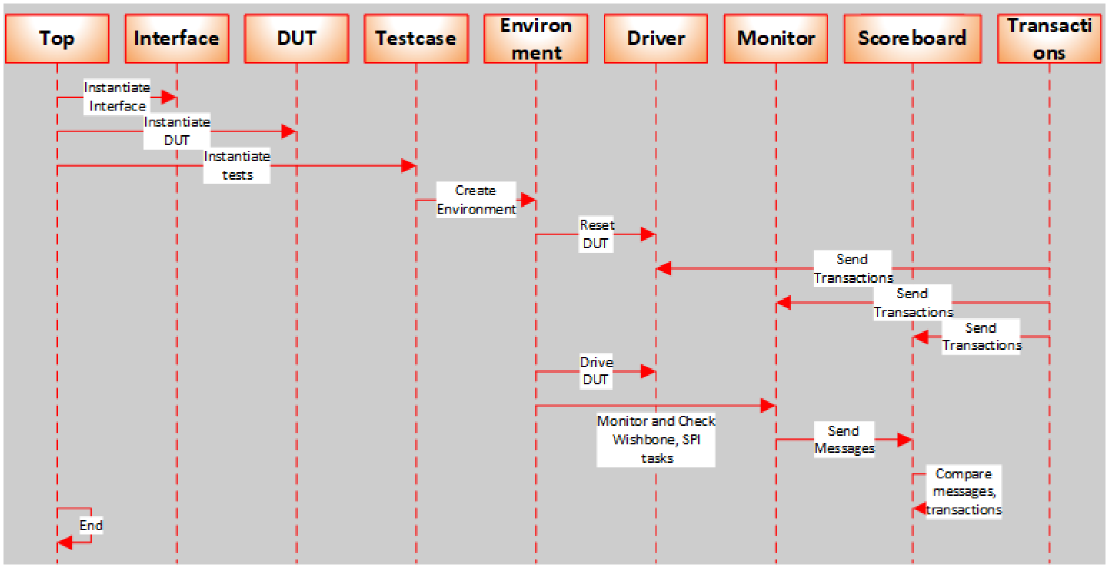


Figure 5.4: Code Flow

5.3.3 Wishbone and SPI Transaction Waveforms

5.3.3.1 Wishbone Transactions to SPI devices

The payload, the data and the type of the transaction is driven using the wishbone read and write tasks in the driver class of the environment in the Fig.5.5. This information is driven to the SPI master and slave devices through the interface signals. The communication starts with the assertion of the wishbone acknowledge (wb_ack_o) and it indicates that the slave is ready. This is visible at 240ns in the Fig. 5.5. At this time frame, The wishbone address (wb_adr_i) is selected as 04 and this indicates the selection of the transmit-receive registers. Wishbone cycle and strobe are set high, indicated by signals wb_cyc_i and wb_stb_i. Write-Enable (wb_we_i) is set low indicating a read operation from the SPI. The Wishbone Select (wb_sel_i) is always 'FFFF' during a read or write operation as only SPI is considered as a slave. A Wishbone address "h 0c' indicates selection of transmit-receive register, "h 14' indicates the SPI divider select value in the SPI register, "h 18' indicates the slave select in the SPI register, and "h 10' indicates the SPI control in the SPI control register. When the write-enable (wb_we_i) is asserted, it indicates a write operation to the SPI master device. This can be seen between 420ns and 460ns in the Fig. 5.5.

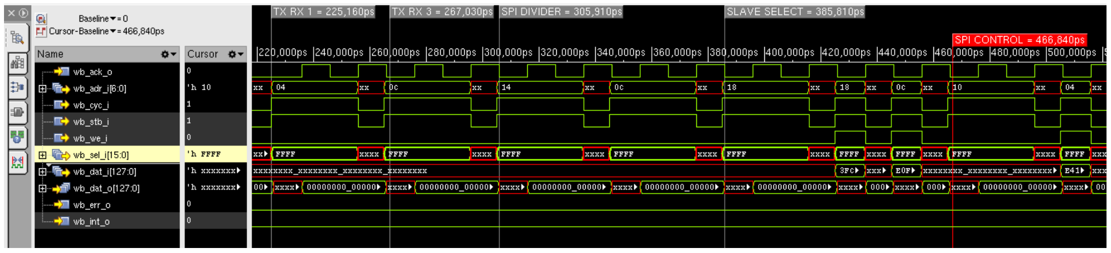


Figure 5.5: Wishbone to SPI Master Transactions

5.3.3.2 SPI Master-Slave Communication

The Fig. 5.6 indicates the communication between the SPI Master initiation device and the SPI slave peripheral device. The data received from the Wishbone is parallel data, 127 bits wide and is indicated by data[127:0]. The transaction begins with an active low slave select, indicated by ss, and remains low till the transfer of data is complete. 'sck' represents the serial clock and it synchronizes the data on mosi and miso. At 420ns, the mosi is high on the rising edge of the serial clock sck. The data on this line changes on every rising edge of the serial clock sck. It can be seen in Fig. 5.6 that data '0101101' is being sent on mosi with one-bit every clock cycle of sck during the time frame between 400ns and 540ns. It can also be seen that the slave device sends all zeros during this time frame.

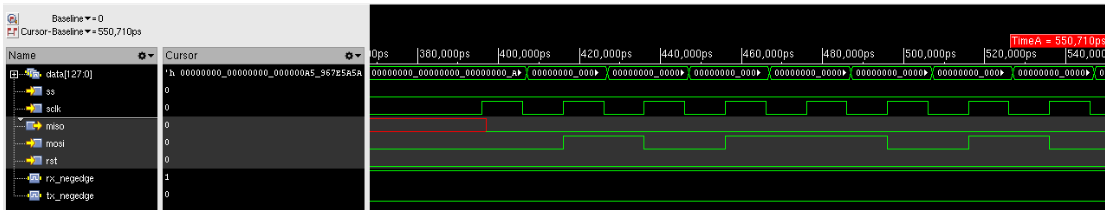


Figure 5.6: SPI Master and Slave Communication

Fig. 5.7 explicitly shows the selection of the slave device through the ss_o signal and the transfer of data from the SPI master device to slave device through the mosi_o. This snapshot was captured during the monitoring of the SPI signals by the monitor class of the environment.

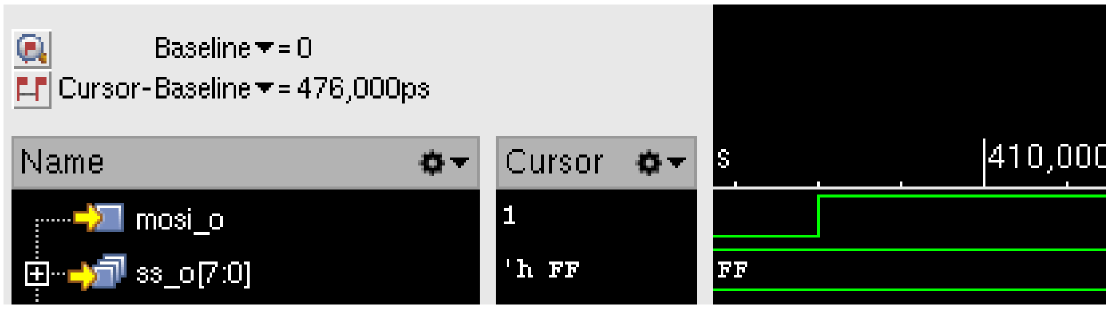


Figure 5.7: SPI Monitor Signals

Chapter 6

Conclusion

A configurable SystemVerilog verification environment for a Wishbone-compliant Serial Peripheral Interface Communication protocol is developed in this work, and this verification environment is used for the validation of parallel to serial to parallel communication on the chip. All verification system components are configured using an input configuration control file. The verification environment extensively validates the full-duplex data transfer between the SPI master and slave devices for different character lengths and different transfer formats. It enables a robust monitoring environment to validate the cycle by cycle operation.

The test bench can be configured to suit different protocol characteristics such as data bus width, address bus width, character length, and type of transmission. The major test bench components used to build the test bench framework are the interface, a Wishbone driver, an environment, a test case and a top-level test module. The Interface connects the host controller and the design under test with the test bench, the driver is in the form of Wishbone read and write cycles to drive the design, the Environment encapsulates the driver and test case scenarios and is responsible for the operational flow at the top-level.

6.1 Future Work

This research has a lot of scope for future work and a few ideas that could be developed include:

- It can be extended to the verification of the serial peripheral interface with a **F**irst-**I**n-**F**irst-**O**ut (**FIFO**) principle.

- Code coverage with respect to the standards can also be a part of the environment.
- The development of verification environment can be extended to the verification other Wishbone-compliant peripherals that support additional communication protocols.

References

[1] O. Cores, *Wishbone B4 System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores*, b4 spec ed., Open Cores, 2010.

[2] R. Chen, S.-z. Huang, W. Lin, and L. Li, "Design and implementation of a reused interface," in *Information Science and Engineering (ICISE), 2009 1st International Conference on*. IEEE, 2009, pp. 2603–2605.

[3] V. R. T. Deep and S. Sujatha, "Synthesis using 65nm library of spi master-slave with wishbone interface," in *International Journal of Advanced Computer Engineering and Communication Technology*, ser. 2, vol. 2, no. ISSN (Print) 2278-5140, 2013, pp. 12–16.

[4] N. Gopal, "Spi controller core: Verification," in *SSRG International Journal of VLSI & Signal Processing (SSRG-IJVSP) Volume 2 Issue 5*. International Journals SSRG, 2015, pp. 1–7.

[5] S. Srot, *SPI Master Core Specification*, rev. 0.6 ed., Open Cores, 03 2004.

[6] M. Keaveney, A. McMahon, N. O'Keeffe, K. Keane, and J. O'Reilly, "The development of advanced verification environments using system verilog," *IET Digital Library*, 2008.

[7] C. Browy, G. Gullikson, and M. Indovina, "A top-down approach to ic design," *Integrated Circuit Design Methodology Guide*, 1997.

[8] D. Ahlawat and N. K. Shukla, "Performance analysis of verilog directed testbench vs constrained random systemverilog testbench," *International Journal of Computer Applications*, vol. 118, no. 22, 2015.

[9] A. Ismail and H. Saafan, "Synthesizable system verilog model for hardware metastability in formal verification," in *Electronics, Communications and Photonics Conference (SIECPC), 2013 Saudi International*. IEEE, 2013, pp. 1–6.

[10] Y. Li, W. Wu, L. Hou, and H. Cheng, “A study on the assertion-based verification of digital ic,” in *Information and Computing Science, 2009. ICIC’09. Second International Conference on*, vol. 2. IEEE, 2009, pp. 25–28.

[11] Y. Tao, “An introduction to assertion-based verification,” in *ASIC, 2009. ASICON’09. IEEE 8th International Conference on.* IEEE, 2009, pp. 1318–1323.

[12] M.-K. You and G.-Y. Song, “Case study: Co-simulation and co-emulation environments based on systemc & systemverilog,” in *TENCON 2009-2009 IEEE Region 10 Conference.* IEEE, 2009, pp. 1–4.

[13] M.-K. You, Y.-J. Oh, and G.-Y. Song, “Implementation of a hardware functional verification system using systemc infrastructure,” in *TENCON 2009-2009 IEEE Region 10 Conference.* IEEE, 2009, pp. 1–5.

[14] C. Spear, *SystemVerilog for verification: a guide to learning the testbench language features.* Springer Science & Business Media, 2008.

[15] Z. Zhou, Z. Xie, X. Wang, and T. Wang, “Development of verification envioronment for spi master interface using systemverilog,” in *Signal Processing (ICSP), 2012 IEEE 11th International Conference on*, vol. 3. IEEE, 2012, pp. 2188–2192.

[16] S. Cohen and L. M. Northrop, “Object-oriented technology and domain analysis,” in *Software Reuse, 1998. Proceedings. Fifth International Conference on.* IEEE, 1998, pp. 86–93.

[17] D. Ahlawat and N. K. Shukla, “Dut verification through an efficient and reusable environment with optimum assertion and functional coverage in systemverilog,” *International Journal of Advanced Computer Science and Applications*, vol. 5, no. 4, 2014.

[18] D. A. S. Committee *et al.*, “Ieee standard for systemverilog unified hardware design, specification, and verification language standard ieee 1800,” *http://www. edastds. org/sv/*, 2012.

[19] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Springer Science & Business Media, 2006.

[20] A. Oudjida, M. Berrandjia, A. Liacha, R. Tiar, K. Tahraoui, and Y. Alhoumays, "Design and test of general-purpose spi master/slave ips on opb bus," in *Systems Signals and Devices (SSD), 2010 7th International Multi-Conference on.* IEEE, 2010, pp. 1–6.

[21] F. Leens, "An introduction to spi and i2c protocol," *IEEE Instrumentation and Measurement magazine*, 2009.

[22] G. HARIKA and G. R. KISHORE, "Development of reusable verification environment for spi protocol," *International Journal of Scientific Engineering and Technology Research*, 2015.

[23] F. Semiconductor, *M68HC11 Reference Manual*, Semiconductor, Freescale, 2007.

[24] J. K. Singh, M. Tiwari, V. Sharma, and S. S. M. INDIA, "Implementation of spi–slave on fpga," *International Journal of Advanced Engineering Technology*, 2012.

[25] J. Zhang, C. Wu, W. Zhang, and J. Wang, "The design and realization of a comprehensive spi interface controller," in *Mechanic Automation and Control Engineering (MACE), 2011 Second International Conference on.* IEEE, 2011, pp. 4529–4532.

[26] M. Greco, M. P. Bussa, L. Ferrero, M. Maggiora, and A. Verna, "Vhdl implementation of a spi controller for panda digital signal processing," in *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2011 IEEE.* IEEE, 2011, pp. 885–888.

[27] K. Aditya, M. Sivakumar, F. Noorbasha, and T. P. Blessington, "Design and functional verification of a spi master slave core using system verilog," *International Journal of Soft Computing and Engineering*, vol. 2, no. 2, pp. 558–563, 2012.

[28] T. Liu and Y. Wang, "Ip design of universal multiple devices spi interface," in *Anti-Counterfeiting, Security and Identification (ASID), 2011 IEEE International Conference on.* IEEE, 2011, pp. 169–172.

[29] N. Anand, G. Joseph, S. S. Oommen, and R. Dhanabal, "Design and implementation of a high speed serial peripheral interface," in *Advances in Electrical Engineering (ICAEE), 2014 International Conference on.* IEEE, 2014, pp. 1–3.

[30] Roopesh, Siddesha, and K. Narayan, “Rtl design and verification of spi master-slave using uvm,” in *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE), August 2015 Volume 4, Issue 8*, ser. 5, vol. 4, no. 2278 - 909X, IJARECE. IJARECE, 08 2015, pp. 1–4.

Appendix I

Source Code

I.1 Transactions

```
class transaction;

  //Declaring the transactions
  rand bit        [32-1:0] addr;
  rand bit       [128-1:0] data;
  rand trans_t             kind;

  constraint addr_range {addr inside  {32'h00000000, 32'h00000004
     , 32'h00000008,
                                       32'h0000000c, 32'h00000010
                                          , 32'h00000014,
                                       32'h00000018}; }

endclass : transaction
```

```
class mntrsbtrans;

  //Declaring the transactions
  rand bit      [32-1:0] addr;
  rand bit     [512-1:0] data;
  rand trans_t           kind;
  rand ctrl_t            ctrl;

  covergroup cg;
    coverpoint addr
    {
            bins valid[] = {SPI_TXRX_0, SPI_TXRX_1, SPI_TXRX_2,
              SPI_TXRX_3,
                            SPI_CTRL, SPI_DIVIDER, SPI_SS};
            illegal_bins invalid = default;
    }
    char_len: coverpoint ctrl.char_len
    {
            bins tiny = { [1:171] };
            bins mid  = { [172:341] };
            bins big  = { 0, [342:511] };
    }
    coverpoint kind;
  endgroup

  function new();
    cg = new();
  endfunction

  function sample_coverage();
```

```
     cg.sample();
  endfunction

endclass : mntrsbtrans
```

I.2 Interface

```
interface dut_intf(input wb_clk_i);

        // Wishbone signals
        //logic                   wb_clk_i;
        logic             wb_rst_i;
        logic [6:0]    wb_adr_i;
        logic [127:0] wb_dat_i;
        logic [127:0] wb_dat_o;
        logic [15:0]   wb_sel_i;
        logic             wb_we_i;
        logic             wb_stb_i;
        logic             wb_cyc_i;
        logic             wb_ack_o;
        logic             wb_err_o;
        logic             wb_int_o;
        logic         test_mode;
        logic         scan_in0;
        logic         scan_en;
        logic         scan_out0;

        // SPI signals
        logic [`SPI_SS_NB-1:0]    ss_o;
        logic                                sclk_o;
        logic                                        mosi_o;
        logic                                        miso_i;

        // Wishbone Signals Modports to be used by the testbench
        modport wb (output wb_adr_i, wb_dat_i, wb_sel_i, wb_we_i,
                                                  wb_stb_i,
                                                     wb_cyc_i,
                                     input wb_dat_o, wb_ack_o,
                                        wb_err_o, wb_int_o);

        // SPI Signals Modports to be used by the testbench
        modport spi (output miso_i, input ss_o, sclk_o, mosi_o);

endinterface : dut_intf
```

I.3 Driver

```
class driver;
  virtual dut_intf dut_if;
  mailbox drvr2sb;
  transaction trans;
  //scoreboard sb;

  function new(virtual dut_intf dut_if, mailbox drvr2sb);
    this.dut_if = dut_if;
    this.drvr2sb = drvr2sb;
    //this.sb = sb;
  endfunction



  task reset;
          dut_if.wb_rst_i = 1'b0;
          repeat(2) @(posedge dut_if.wb_clk_i);
          dut_if.wb_rst_i = 1'b1;
    $display("Reset started...");
    dut_if.wb.wb_adr_i <= {32{1'bx}};
    dut_if.wb.wb_dat_i <= {128{1'bx}};
    dut_if.wb.wb_sel_i <= {16{1'bx}};
    dut_if.wb.wb_we_i <= 1'hx;
    dut_if.wb.wb_stb_i <= 1'bx;
    dut_if.wb.wb_cyc_i <= 1'b0;
    $display("Reset ended...");
          repeat(20) @(posedge dut_if.wb_clk_i);
          dut_if.wb_rst_i = 1'b0;
  endtask
```

```
task drive(input integer iteration);

  repeat(iteration)
  begin
              $display("%b", dut_if.wb_clk_i);
    trans = new();
    dut_if.wb.wb_we_i <= 1'b0;
    dut_if.wb.wb_stb_i <= 1'b0;
    dut_if.wb.wb_cyc_i <= 1'b0;

    @ (negedge dut_if.wb_clk_i);
    if(trans.randomize())
    begin
      if(trans.kind == RX)
        wb_read(1, trans.addr, trans.data);
      else
        wb_write(1, trans.addr, trans.data);

      if((trans.addr == SPI_CTRL) && (trans.data & SPI_GO))
        @(posedge dut_if.wb.wb_int_o);

      drvr2sb.put(trans);
    end

  end
endtask


task automatic wb_write;
  input delay;
  integer delay;
```

```
    input [32-1:0] g_address;
    input [128-1:0] g_data;

    begin
    repeat(delay)
        @(posedge dut_if.wb_clk_i);
                    dut_if.wb.wb_adr_i = g_address;
                    dut_if.wb.wb_dat_i = g_data;
                    dut_if.wb.wb_cyc_i = 1'b1;
                    dut_if.wb.wb_stb_i = 1'b1;
                    dut_if.wb.wb_we_i = 1'b1;
                    dut_if.wb.wb_sel_i = {16{1'b1}};
                    @(posedge dut_if.wb_clk_i);
        #1;

                    //Waiting for acknowledge from slave
                    while(~dut_if.wb.wb_ack_o) begin
                           @(posedge dut_if.wb_clk_i);
                           #1;
                    end

                    dut_if.wb.wb_cyc_i = 1'b0;
                    dut_if.wb.wb_stb_i = 1'bx;
                    dut_if.wb.wb_we_i = 1'hx;
                    dut_if.wb.wb_sel_i = {16{1'bx}};
                    dut_if.wb.wb_adr_i = {32{1'bx}};
                    dut_if.wb.wb_dat_i = {128{1'bx}};
    end
  endtask



  task automatic wb_read;
     input delay;
```

```
          integer delay;

          input [32-1:0] g_address;
          output [128-1:0] g_data;

          begin
                  repeat(delay)
        @(posedge dut_if.wb_clk_i);
                  dut_if.wb.wb_adr_i = g_address;
                  dut_if.wb.wb_dat_i = {128{1'bx}};
                  dut_if.wb.wb_cyc_i = 1'b1;
                  dut_if.wb.wb_stb_i = 1'b1;
                  dut_if.wb.wb_we_i = 1'b0;
                  dut_if.wb.wb_sel_i = {16{1'b1}};
                  @(posedge dut_if.wb_clk_i);

                  //Waiting for acknowledge from slave
                  while(~dut_if.wb.wb_ack_o) begin
                         @(posedge dut_if.wb_clk_i);
                         #1;
                  end

                  dut_if.wb.wb_cyc_i = 1'b0;
                  dut_if.wb.wb_stb_i = 1'bx;
                  dut_if.wb.wb_adr_i = {32{1'bx}};
                  dut_if.wb.wb_dat_i = {128{1'bx}};
                  dut_if.wb.wb_we_i = 1'hx;
                  dut_if.wb.wb_sel_i = {16{1'bx}};
                  g_data = dut_if.wb.wb_dat_o;

          end
  endtask

endclass : driver
```

I.4 Scoreboard

```
class scoreboard;

  mailbox drvr2sb;
  mailbox mntr2sb;


  function new(mailbox mntr2sb,mailbox drvr2sb);
    this.drvr2sb = drvr2sb;
    this.mntr2sb = mntr2sb;
  endfunction

  task main;
    mntrsbtrans trans_rcv, trans_exp;
    repeat(4) begin
      mntr2sb.get(trans_rcv);
      $display("Monitor Transaction received");
      //$display("Monitor Address: %0d, Data: %0d\n", trans.addr,
          trans.data);

      drvr2sb.get(trans_exp);
      $display("Driver Transaction received");
      //$display("Driver Address: %0d, Data: %0d\n", trans.addr,
         trans.data);
      //if(trans_rcv.compare(trans_exp))
                  //$display("%0d: Transaction matched", $time);
      //else
        //$root.error++;


      //for (int i = 0; i < (trans.ctrl & 'h3f); i++)
```

```
        //mask[i] = 1;

      //if ((actual.data & mask) !== (expected.data & mask));
        //$display("Compare Failed");
    end
  endtask : main
endclass : scoreboard
```

I.5 Monitor

```
class monitor;
  virtual dut_intf dut_if;

  mailbox mntr2sb;

  function new(virtual dut_intf dut_if, mailbox mntr2sb);
    this.dut_if = dut_if;
    this.mntr2sb = mntr2sb;
  endfunction

  task wbmon_check;
    forever begin
      mntrsbtrans trans;
      trans = new();

      @(posedge dut_if.wb_cyc_i);
      while(~dut_if.wb_stb_i) @(posedge dut_if.wb_clk_i);
        while(~dut_if.wb_ack_o) @(posedge dut_if.wb_clk_i);

      while (dut_if.wb_ack_o) begin : WB_ACK
        logic [127:0] tmp_dat;

        if (dut_if.wb_cyc_i !== 1 || dut_if.wb_stb_i !== 1)
          break;
        if (dut_if.wb_we_i === 1'b1) begin
          $display("Wishbone Write Enable is : %v", dut_if.
             wb_we_i);
          trans.kind = TX;
          tmp_dat = dut_if.wb_dat_i;
```

```
end else begin
  $display("Wishbone Write Enable is : %v", dut_if.
     wb_we_i);
  trans.kind = RX;
  tmp_dat = dut_if.wb_dat_o;
end

trans.addr = dut_if.wb_adr_i;
case (trans.addr)
  SPI_TXRX_0: trans.data[127:0]   = tmp_dat;
  SPI_TXRX_1: trans.data[255:128] = tmp_dat;
  SPI_TXRX_2: trans.data[383:256] = tmp_dat;
  SPI_TXRX_3: trans.data[511:384] = tmp_dat;
  SPI_SS: ;
  SPI_DIVIDER: ;
  SPI_CTRL: trans.ctrl = tmp_dat;
  default: trans.addr = 'h1f;                    //Invalid
       Address
endcase

if (trans.addr != 'h1f) begin
  $display("Address = %0x, Data = %0x", trans.addr,
     tmp_dat);
end else
  $display("Address = %0x, Data = %0x", trans.addr,
     tmp_dat);

if ((trans.kind == TX) && (trans.addr == SPI_CTRL) && (
   trans.ctrl & SPI_GO))
begin
    trans.sample_coverage();
    mntr2sb.put(trans);

    trans = new();
```

```
        end

        @(posedge dut_if.wb_clk_i);
      end : WB_ACK
    end
  endtask : wbmon_check

  task spimon_check;
    forever begin
      bit [6:0] i = 0;

      mntrsbtrans trans = new();

      while (dut_if.ss_o)
      begin
        for (i=0; dut_if.ss_o[0]; i++)
          begin
            @(posedge dut_if.sclk_o);
            trans.data[i] = dut_if.spi.mosi_o;
          end
      end

      if (i) begin
        mntr2sb.put(trans);
      end
    end
  endtask : spimon_check

  task main();
    fork
      wbmon_check;
      spimon_check;
    join_none
  endtask : main
```

```
endclass : monitor
```

I.6 Environment

```
class environment;

  driver driv;
  monitor mon;
  scoreboard sb;

  mailbox mntr2sb, drvr2sb;

  virtual dut_intf intf;

  function new (virtual dut_intf intf);
    this.intf = intf;
    this.drvr2sb = drvr2sb;
    this.mntr2sb = mntr2sb;
    driv = new(intf, drvr2sb);
    mon = new(intf, mntr2sb);
    sb = new(mntr2sb, drvr2sb);
  endfunction

endclass : environment
```

I.7 Testcase

```
program testcase(dut_intf intf);

  environment env = new(intf);

  initial
  begin
        $timeformat(-9, 2, "ns", 16);
    env.driv.reset();
    env.driv.drive(10);
    env.mon.main();
    env.sb.main();
  end

endprogram : testcase
```

I.8 Test

```
module test();

  logic wb_clk_i = 0;

  initial
  forever
      #5 wb_clk_i = ~wb_clk_i;


  dut_intf intf(wb_clk_i);

  as_spi top (intf.wb_rst_i, wb_clk_i, intf.scan_in0, intf.
     scan_en,
                intf.test_mode,intf.scan_out0, intf.wb.wb_adr_i,
                   intf.wb.wb_dat_i,
                intf.wb.wb_dat_o, intf.wb.wb_sel_i, intf.wb.wb_we_i
                   ,
                intf.wb.wb_stb_i, intf.wb.wb_cyc_i, intf.wb.
                   wb_ack_o,
                intf.wb.wb_err_o, intf.wb.wb_int_o, intf.spi.ss_o,
                intf.spi.sclk_o, intf.spi.mosi_o, intf.spi.miso_i);

  as_spi_slave SPI_SLAVE (1'b0, intf.spi.ss_o[0], intf.spi.sclk_o
     ,
                             intf.spi.mosi_o, intf.spi.miso_i);

  testcase t1(intf);

endmodule : test
```

I.9 Design Under Test

I.9.1 Clock Generator

```
`include "src/as_spi_defines.v"
`include "src/timescale.v"


module as_spi_clkgen
                (
      input rst,
                                              //System reset
      input in_clk,
                                      //System Clock
                input in_clk_en,
                                                                //Clock
                   Enable
                input go,
                                                                        //
                   Start transfer
                input last_clk,
                                                //Last Clock
                input [`SPI_DIVIDER_LEN-1:0] clk_divider,//Clock
                   divider
                output reg out_clk,
                                                //Output Clock
                output reg out_clk_pos_edge,
                                //Output Clock positive edge pulse
                output reg out_clk_neg_edge
                                //Output Clock negative edge pulse
                );
```

```
reg     [`SPI_DIVIDER_LEN-1:0]  cnt;
     //Clock Counter
wire
                     cnt_zero;
wire
                     cnt_one;


assign cnt_zero = cnt == {`SPI_DIVIDER_LEN{1'b0
   }};
assign cnt_one = cnt == {{`SPI_DIVIDER_LEN-1{1'b0
   }}, 1'b1};


//Clock counter counting half period
always @(posedge in_clk or posedge rst) begin
  if (rst)
    cnt <= #1 {`SPI_DIVIDER_LEN{1'b1}};
  else begin
    if (!in_clk_en || cnt_zero)
      cnt <= #1 clk_divider;
         else
               cnt <= #1 cnt - {{
                  `SPI_DIVIDER_LEN-1{1'b0}}, 1'
                  b1};
  end
end


//Asserting output clock every other half period
always @(posedge in_clk or posedge rst) begin
  if(rst)
    out_clk <= #1 1'b0;
  else begin
```

```
                if (in_clk_en && cnt_zero && (out_clk ||
                    !last_clk))
                  out_clk <= #1 ~out_clk;
                else
                  out_clk <= #1 out_clk;
          end
        end


        //The following always block is for the posedge
           and negedge pulses of the clock
        always @(posedge in_clk or posedge rst) begin
          if (rst) begin
            out_clk_pos_edge <= 1'b0;
                  out_clk_neg_edge <= 1'b0;
          end else begin
            out_clk_pos_edge <= ((in_clk_en && !out_clk
               && cnt_one) || (!(|clk_divider) && out_clk
               ) || (!(|clk_divider) && go && !in_clk_en)
               );
                  out_clk_neg_edge <= ((in_clk_en &&
                     out_clk && cnt_one) || (!(|
                     clk_divider) && !out_clk && in_clk_en
                     ));
          end
        end

endmodule
```

I.9.2 Shift Register

```
`include "src/as_spi_defines.v"
`include "src/timescale.v"

module as_spi_shift
                (
      input rst,
                                               //System reset
      input clk,
                                               //System Clock
                input [3:0] latch,
                                                 //Stores the data in
                    the shift register
                input [15:0] byte_sel,
                                          //Stores the byte selection
                   signals
                input [`SPI_CHAR_LEN_BITS-1:0] len,
                     //Stores the length of the data in bits
                input lsb,
                                                               //
                   least significant bit first
                input go,
                                                               //
                   Indicates start of transfer
                input pos_edge,
                                                 //Indicates the
                   recognition of posedge of sclk
                input neg_edge,
                                                        //Indicates
                   the recognition of negedge of the sclk
```

```
        input rx_negedge,
                                            //Serial Input
          sampled on negedge
        input tx_negedge,
                                            //Serial Output
          driven on negedge
        output reg tip,
                                            //Indicates the
          transfer is in progress
    output last,
                                        //Indicates the last bit
      of transfer
        input [127:0] p_in,
                                            //Indicates the
          input parallel data
        output [`SPI_MAX_CHAR-1:0] p_out,
            //Indicates the output parallel data
        input s_clk,
                                                  //Indicates
          the serial input clock
        input s_in,
                                                  //Indicates
          the serial input data
        output reg s_out
                                                  //Indicates
          the serial output data
        );

        reg [`SPI_CHAR_LEN_BITS:0] cnt;
            //Data bit Counter
        reg [`SPI_MAX_CHAR-1:0] data;
                  //Shift Register
        wire [`SPI_CHAR_LEN_BITS:0] tx_bit_pos;  //Next
          bit position on the Tx side
```

```
wire [`SPI_CHAR_LEN_BITS:0] rx_bit_pos;  //Next
   bit position on the Rx side
wire rx_clk;
                                        //Rx Clock
   Enable
wire tx_clk;
                                        //Tx Clock
   Enable

assign p_out = data;

assign tx_bit_pos = lsb ? {!(|len), len} - cnt :
   cnt - {{`SPI_CHAR_LEN_BITS{1'b0}},1'b1};

assign rx_bit_pos = lsb ? {!(|len), len} - (
   rx_negedge ? cnt + {{`SPI_CHAR_LEN_BITS{1'b0
   }},1'b1} : cnt) :
```

```
assign last = !(|cnt);

assign rx_clk = (rx_negedge ? neg_edge : pos_edge
   ) && (!last || s_clk);
assign tx_clk = (tx_negedge ? neg_edge : pos_edge
   ) && !last;

// Counter for Character bits
always @(posedge clk or posedge rst) begin
  if (rst)
    cnt <= #1 {`SPI_CHAR_LEN_BITS+1{1'b0}};
  else begin
         if (tip)
           cnt <= #1 pos_edge ? (cnt - {{
              `SPI_CHAR_LEN_BITS{1'b0}}, 1'b1}) :
                cnt;
         else
           cnt <= #1 !(|len) ? {1'b1, {
              `SPI_CHAR_LEN_BITS{1'b0}}} : {1'b0,
                len};
  end
end

//When the transfer of data is in progress
always @(posedge clk or posedge rst) begin
  if (rst)
    tip <= #1 1'b0;
  else if(go && ~tip)
```

```
            tip <= #1 1'b1;
    else if(tip && last && pos_edge)
            tip <= #1 1'b0;
end

//Transmitting bits to the line or bus
always @(posedge clk or posedge rst) begin
  if (rst)
            s_out <= #1 1'b0;
  else
            s_out <= #1 (tx_clk || !tip) ? data[
               tx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]] :
                s_out;
end

//Receiving bits from the line or bus
always @(posedge clk or posedge rst) begin
  if (rst)
            data <= #1 {`SPI_MAX_CHAR{1'b0}};
  `ifdef SPI_MAX_CHAR_512
  else if (latch[0] && !tip) begin
                    if(byte_sel[15])
                            data[127:120] <= #1 p_in
                               [127:120];
                    if(byte_sel[14])
                            data[119:112] <= #1 p_in
                               [119:112];
                    if(byte_sel[13])
                            data[111:104] <= #1 p_in
                               [111:104];
                    if(byte_sel[12])
                            data[103:96] <= #1 p_in
                               [103:96];
                    if(byte_sel[11])
                data[95:88] <= #1 p_in
                    [95:88];
            if(byte_sel[10])
                data[87:80] <= #1 p_in
                    [87:80];
            if(byte_sel[9])
                data[79:72] <= #1 p_in
                    [79:72];
            if(byte_sel[8])
                data[71:64] <= #1 p_in
                    [71:64];
            if(byte_sel[7])
                data[63:56] <= #1 p_in
                    [63:56];
            if(byte_sel[6])
                data[57:48] <= #1 p_in
                    [57:48];
            if(byte_sel[5])
                data[47:40] <= #1 p_in
                    [47:40];
            if(byte_sel[4])
                data[39:32] <= #1 p_in
                    [39:32];
            if(byte_sel[3])
                data[31:24] <= #1 p_in
                    [31:24];
            if(byte_sel[2])
                data[23:16] <= #1 p_in
                    [23:16];
            if(byte_sel[1])
                data[15:8] <= #1 p_in
                    [15:8];
            if(byte_sel[0])
```

```
                                        data[7:0] <= #1 p_in
                                           [7:0];
                        end else if (latch[1] && !tip) begin
                                if(byte_sel[15])
                                        data[255:248] <= #1 p_in
                                           [127:120];
                                if(byte_sel[14])
                                        data[247:240] <= #1 p_in
                                           [119:112];
                                if(byte_sel[13])
                                        data[239:232] <= #1 p_in
                                           [111:104];
                                if(byte_sel[12])
                                        data[231:224] <= #1 p_in
                                           [103:96];
                                if(byte_sel[11])
                                        data[223:216] <= #1 p_in
                                           [95:88];
                                if(byte_sel[10])
                                        data[215:208] <= #1 p_in
                                           [87:80];
                                if(byte_sel[9])
                                        data[207:200] <= #1 p_in
                                           [79:72];
                                if(byte_sel[8])
                                        data[199:192] <= #1 p_in
                                           [71:64];
                                if(byte_sel[7])
                                        data[191:184] <= #1 p_in
                                           [63:56];
                                if(byte_sel[6])
                                        data[183:176] <= #1 p_in
                                           [57:48];
                                if(byte_sel[5])
```

```
                                        data[175:168] <= #1 p_in
                                            [47:40];
                                if(byte_sel[4])
                                        data[167:160] <= #1 p_in
                                            [39:32];
                                if(byte_sel[3])
                                        data[159:152] <= #1 p_in
                                            [31:24];
                                if(byte_sel[2])
                                        data[151:144] <= #1 p_in
                                            [23:16];
                                if(byte_sel[1])
                                        data[143:136] <= #1 p_in
                                            [15:8];
                                if(byte_sel[0])
                                        data[135:128] <= #1 p_in
                                            [7:0];
                        end else if (latch[2] && !tip) begin
                                if(byte_sel[15])
                                        data[383:376] <= #1 p_in
                                            [127:120];
                                if(byte_sel[14])
                                        data[375:368] <= #1 p_in
                                            [119:112];
                                if(byte_sel[13])
                                        data[367:360] <= #1 p_in
                                            [111:104];
                                if(byte_sel[12])
                                        data[359:352] <= #1 p_in
                                            [103:96];
                                if(byte_sel[11])
                                        data[351:344] <= #1 p_in
                                            [95:88];
                                if(byte_sel[10])
                                        data[343:336] <= #1 p_in
                                            [87:80];
                                if(byte_sel[9])
                                        data[335:328] <= #1 p_in
                                            [79:72];
                                if(byte_sel[8])
                                        data[327:320] <= #1 p_in
                                            [71:64];
                                if(byte_sel[7])
                                        data[319:312] <= #1 p_in
                                            [63:56];
                                if(byte_sel[6])
                                        data[311:304] <= #1 p_in
                                            [57:48];
                                if(byte_sel[5])
                                        data[303:296] <= #1 p_in
                                            [47:40];
                                if(byte_sel[4])
                                        data[295:288] <= #1 p_in
                                            [39:32];
                                if(byte_sel[3])
                                        data[287:280] <= #1 p_in
                                            [31:24];
                                if(byte_sel[2])
                                        data[279:272] <= #1 p_in
                                            [23:16];
                                if(byte_sel[1])
                                        data[271:264] <= #1 p_in
                                            [15:8];
                                if(byte_sel[0])
                                        data[263:256] <= #1 p_in
                                            [7:0];
                        end else if (latch[3] && !tip) begin
                                if(byte_sel[15])
```

```
                                        data[511:504] <= #1 p_in
                                            [127:120];
                                if(byte_sel[14])
                                        data[503:496] <= #1 p_in
                                            [119:112];
                                if(byte_sel[13])
                                        data[495:488] <= #1 p_in
                                            [111:104];
                                if(byte_sel[12])
                                        data[487:480] <= #1 p_in
                                            [103:96];
                                if(byte_sel[11])
                                        data[479:472] <= #1 p_in
                                            [95:88];
                                if(byte_sel[10])
                                        data[471:464] <= #1 p_in
                                            [87:80];
                                if(byte_sel[9])
                                        data[463:456] <= #1 p_in
                                            [79:72];
                                if(byte_sel[8])
                                        data[455:448] <= #1 p_in
                                            [71:64];
                                if(byte_sel[7])
                                        data[447:440] <= #1 p_in
                                            [63:56];
                                if(byte_sel[6])
                                        data[439:432] <= #1 p_in
                                            [57:48];
                                if(byte_sel[5])
                                        data[431:424] <= #1 p_in
                                            [47:40];
                                if(byte_sel[4])
```

```
                                        data[423:416] <= #1 p_in
                                           [39:32];
                                  if(byte_sel[3])
                                        data[415:408] <= #1 p_in
                                           [31:24];
                                  if(byte_sel[2])
                                        data[407:400] <= #1 p_in
                                           [23:16];
                                  if(byte_sel[1])
                                        data[399:392] <= #1 p_in
                                           [15:8];
                                  if(byte_sel[0])
                                        data[391:384] <= #1 p_in
                                           [7:0];
                         end
              `else
              `ifdef SPI_MAX_CHAR_256
                else if (latch[0] && !tip) begin
                        if(byte_sel[15])
                          data[127:120] <= #1 p_in[127:120];
                        if(byte_sel[14])
                          data[119:112] <= #1 p_in[119:112];
                        if(byte_sel[13])
                          data[111:104] <= #1 p_in[111:104];
                        if(byte_sel[12])
                          data[103:96] <= #1 p_in[103:96];
                        if(byte_sel[11])
                          data[95:88] <= #1 p_in[95:88];
                        if(byte_sel[10])
                          data[87:80] <= #1 p_in[87:80];
                        if(byte_sel[9])
                          data[79:72] <= #1 p_in[79:72];
                        if(byte_sel[8])
                          data[71:64] <= #1 p_in[71:64];
```

```
                    if(byte_sel[7])
                      data[63:56] <= #1 p_in[63:56];
                    if(byte_sel[6])
                      data[57:48] <= #1 p_in[57:48];
                    if(byte_sel[5])
                      data[47:40] <= #1 p_in[47:40];
                    if(byte_sel[4])
                      data[39:32] <= #1 p_in[39:32];
                    if(byte_sel[3])
                      data[31:24] <= #1 p_in[31:24];
                    if(byte_sel[2])
                      data[23:16] <= #1 p_in[23:16];
                    if(byte_sel[1])
                      data[15:8] <= #1 p_in[15:8];
                    if(byte_sel[0])
                      data[7:0] <= #1 p_in[7:0];
                end else if (latch[1] && !tip) begin
                    if(byte_sel[15])
                      data[255:248] <= #1 p_in[127:120];
                    if(byte_sel[14])
                      data[247:240] <= #1 p_in[119:112];
                    if(byte_sel[13])
                      data[239:232] <= #1 p_in[111:104];
                    if(byte_sel[12])
                      data[231:224] <= #1 p_in[103:96];
                    if(byte_sel[11])
                      data[223:216] <= #1 p_in[95:88];
                    if(byte_sel[10])
                      data[215:208] <= #1 p_in[87:80];
                    if(byte_sel[9])
                      data[207:200] <= #1 p_in[79:72];
                    if(byte_sel[8])
                      data[199:192] <= #1 p_in[71:64];
                    if(byte_sel[7])
                      data[191:184] <= #1 p_in[63:56];
                    if(byte_sel[6])
                      data[183:176] <= #1 p_in[57:48];
                    if(byte_sel[5])
                      data[175:168] <= #1 p_in[47:40];
                    if(byte_sel[4])
                      data[167:160] <= #1 p_in[39:32];
                    if(byte_sel[3])
                      data[159:152] <= #1 p_in[31:24];
                    if(byte_sel[2])
                      data[151:144] <= #1 p_in[23:16];
                    if(byte_sel[1])
                      data[143:136] <= #1 p_in[15:8];
                    if(byte_sel[0])
                      data[135:128] <= #1 p_in[7:0];
                end
              `else
                else if (latch[0] && !tip) begin
                        `ifdef SPI_MAX_CHAR_8
                                if (byte_sel[0])
                                        data[7:0]  <= #1 p_in
                                           [7:0];
                        `endif
                        `ifdef SPI_MAX_CHAR_16
                                if (byte_sel[0])
                                        data[7:0]  <= #1 p_in
                                           [7:0];
                                if (byte_sel[1])
                                        data[15:8] <= #1 p_in
                                           [15:8];
                        `endif
                        `ifdef SPI_MAX_CHAR_24
                                if (byte_sel[0])
```

```
                                        data [7:0]    <= #1 p_in
                                           [7:0];
                                 if (byte_sel[1])
                                        data [15:8]   <= #1 p_in
                                           [15:8];
                                 if (byte_sel[2])
                                        data [23:16] <= #1 p_in
                                           [23:16];
                       `endif
                       `ifdef SPI_MAX_CHAR_32
                                 if (byte_sel[0])
                                        data [7:0]    <= #1 p_in
                                           [7:0];
                                 if (byte_sel[1])
                                        data [15:8]   <= #1 p_in
                                           [15:8];
                                 if (byte_sel[2])
                                        data [23:16] <= #1 p_in
                                           [23:16];
                                 if (byte_sel[3])
                                        data [31:24] <= #1 p_in
                                           [31:24];
                       `endif
                       `ifdef SPI_MAX_CHAR_64
                                 if (byte_sel[0])
                                        data [7:0]    <= #1 p_in
                                           [7:0];
                                 if (byte_sel[1])
                                        data [15:8]   <= #1 p_in
                                           [15:8];
                                 if (byte_sel[2])
                                        data [23:16] <= #1 p_in
                                           [23:16];
                                 if (byte_sel[3])
```

```
                        data[31:24] <= #1 p_in
                            [31:24];
                if (byte_sel[4])
                        data[39:32] <= #1 p_in
                            [39:32];
                if (byte_sel[5])
                        data[47:40] <= #1 p_in
                            [47:40];
                if (byte_sel[6])
                        data[55:48] <= #1 p_in
                            [55:48];
                if (byte_sel[7])
                        data[63:56] <= #1 p_in
                            [63:56];
        `endif
        `ifdef SPI_MAX_CHAR_128
                if (byte_sel[0])
                        data[7:0]   <= #1 p_in
                            [7:0];
                if (byte_sel[1])
                        data[15:8]  <= #1 p_in
                            [15:8];
                if (byte_sel[2])
                        data[23:16] <= #1 p_in
                            [23:16];
                if (byte_sel[3])
                        data[31:24] <= #1 p_in
                            [31:24];
                if (byte_sel[4])
                        data[39:32] <= #1 p_in
                            [39:32];
                if (byte_sel[5])
                        data[47:40] <= #1 p_in
                            [47:40];
```

```
                                    if (byte_sel[6])
                                            data[55:48] <= #1 p_in
                                                [55:48];
                                    if (byte_sel[7])
                                            data[63:56] <= #1 p_in
                                                [63:56];
                                    if (byte_sel[8])
                                            data[71:64] <= #1 p_in
                                                [71:64];
                                    if (byte_sel[9])
                                            data[79:72] <= #1 p_in
                                                [79:72];
                                    if (byte_sel[10])
                                            data[87:80] <= #1 p_in
                                                [87:80];
                                    if (byte_sel[11])
                                            data[95:88] <= #1 p_in
                                                [95:88];
                                    if (byte_sel[12])
                                            data[103:96] <= #1 p_in
                                                [103:96];
                                    if (byte_sel[13])
                                            data[111:104] <= #1 p_in
                                                [111:104];
                                    if (byte_sel[14])
                                            data[119:112] <= #1 p_in
                                                [119:112];
                                    if (byte_sel[15])
                                            data[127:120] <= #1 p_in
                                                [127:120];
                            `endif
                             end
                    `endif
                    `endif
```

```
                        else
                          data[rx_bit_pos[`SPI_CHAR_LEN_BITS-1:0]] <=
                              #1 rx_clk ? s_in : data[rx_bit_pos[
                              `SPI_CHAR_LEN_BITS-1:0]];
                     end

endmodule
```

I.9.3 SPI Master

```
`include "src/as_spi_defines.v"
`include "src/timescale.v"
//`include "src/as_spi_clkgen.v"
//`include "src/as_spi_shift.v"



module as_spi
                (
       input                               wb_rst_i,
                   //Synchronous active high
       input                               wb_clk_i,
                   //Master clock input

              // Test Scan mode Inputs and Outputs
       input                               scan_in0,
                   //test scan mode data input
       input                               scan_en,
                   //test scan mode enable
       input                               test_mode,
                   //test mode select
       output                              scan_out0,
                   //test scan mode data output

         //Wishbone Slave port signals
              input       [6:0]   wb_adr_i,
                    //Lower address bits
              input       [128-1:0] wb_dat_i,
                 //Input Databus width
```

```
            output reg [128-1:0] wb_dat_o,
               //Output Databus Width
            input       [15:0]   wb_sel_i,
                       //Byte select inputs
            input                          wb_we_i,
                                       //Write Enable Input
            input                          wb_stb_i,
                                       //Strobe input
            input                          wb_cyc_i,
                                       //Cycle Input
            output reg              wb_ack_o,
                              //Bus cycle acknowledge
               output
            output                                   wb_err_o
               ,                              //Termination
               Error Output
            output reg              wb_int_o,
                              //Interrupt enable request
               output

            //SPI interface signals
            output [`SPI_SS_NB-1:0] ss_o,
               //Slave Select
            output
                        sclk_o,                       //
               Serial SPI Clock
            output
                        mosi_o,                       //
               Master Out Slave In
            input
               miso_i                       //Master In Slave
               Output
);
```

```
//Register accesses and definitions
reg [`SPI_DIVIDER_LEN-1:0] divider;
            //Divider Register
reg [`SPI_CTRL_BIT_NB-1:0] ctrl;
                     //Control and status register
reg [`SPI_SS_NB-1:0] ss;
                              //Slave Select
   Register
reg [128-1:0] wb_dat;
                                        //Wishbone
   data out temporary
wire [`SPI_MAX_CHAR-1:0] rx;
                     //Receiver Register
wire rx_negedge;
                                                  //MISO
    sampled on negedge
wire tx_negedge;
                                                  //MOSI
    driven on negedge
wire [`SPI_CHAR_LEN_BITS-1:0] char_len; //Length
   of the Character
wire go;

                     //Go status flag
wire lsb;

            //LSB on first line
wire ie;

                     //Interrupt Enable
wire ass;

            //Automatic Slave Select
```

```
              wire spi_divider_sel;
                                        //SPI divider register
                 Select
              wire [3:0] spi_tx_sel;
                                        //SPI Transmit
                Register Select
              wire spi_ctrl_sel;
                                              //SPI Control
                Register Select
              wire spi_ss_sel;
                                                    //
                Slave Select Register Select
              wire tip;

                      //SPI Transfer in progress
              wire pos_edge;
                                              //Serial Clock
                 positive edge
              wire neg_edge;
                                                    //
                Serial Clock Negative Edge
              wire last_bit;
                                                    //Last
                 character bit


//Decode the address signals
assign spi_divider_sel = wb_cyc_i & wb_stb_i & (wb_adr_i[
  `SPI_OFS_BITS] == `SPI_DEVIDE);
assign spi_ctrl_sel    = wb_cyc_i & wb_stb_i & (wb_adr_i[
  `SPI_OFS_BITS] == `SPI_CTRL);
assign spi_tx_sel[0]   = wb_cyc_i & wb_stb_i & (wb_adr_i[
  `SPI_OFS_BITS] == `SPI_TX_0);
```

```
assign spi_tx_sel[1]    = wb_cyc_i & wb_stb_i & (wb_adr_i[
   `SPI_OFS_BITS] == `SPI_TX_1);
assign spi_tx_sel[2]    = wb_cyc_i & wb_stb_i & (wb_adr_i[
   `SPI_OFS_BITS] == `SPI_TX_2);
assign spi_tx_sel[3]    = wb_cyc_i & wb_stb_i & (wb_adr_i[
   `SPI_OFS_BITS] == `SPI_TX_3);
assign spi_ss_sel       = wb_cyc_i & wb_stb_i & (wb_adr_i[
   `SPI_OFS_BITS] == `SPI_SS);

//Read Data from the Registers
always @(wb_adr_i or rx or ctrl or divider or ss) begin
  case(wb_adr_i[`SPI_OFS_BITS])
        `ifdef SPI_MAX_CHAR_512
          `SPI_RX_0: wb_dat = rx[127:0];
          `SPI_RX_1: wb_dat = rx[255:128];
          `SPI_RX_2: wb_dat = rx[383:256];
          `SPI_RX_3: wb_dat = {{512-`SPI_MAX_CHAR{1'b0}},rx[
             `SPI_MAX_CHAR-1:384]};
        `else
        `ifdef SPI_MAX_CHAR_256
          `SPI_RX_0: wb_dat = rx[127:0];
          `SPI_RX_1: wb_dat = {{256-`SPI_MAX_CHAR{1'b0}},rx[
             `SPI_MAX_CHAR-1:128]};
          `SPI_RX_2: wb_dat = 128'b0;
          `SPI_RX_3: wb_dat = 128'b0;
        `else
          `SPI_RX_0: wb_dat = {{128-`SPI_MAX_CHAR{1'b0}},rx[
             `SPI_MAX_CHAR-1:0]};
          `SPI_RX_1: wb_dat = 128'b0;
          `SPI_RX_2: wb_dat = 128'b0;
          `SPI_RX_3: wb_dat = 128'b0;
        `endif
        `endif
```

```
                        `SPI_CTRL:     wb_dat = {{128-`SPI_CTRL_BIT_NB{1'
                            b0}},ctrl};
                        `SPI_DEVIDE:  wb_dat = {{128-`SPI_DIVIDER_LEN{1'
                            b0}},divider};
                        `SPI_SS:       wb_dat = {{128-`SPI_SS_NB{1'b0}},ss
                            };
                        default:       wb_dat = 128'bx;
   endcase
end

//Wishbone data output
always @(posedge wb_clk_i or posedge wb_rst_i) begin
   if (wb_rst_i)
     wb_dat_o <= #1 128'b0;
   else
     wb_dat_o <= #1 wb_dat;
end

//Wishbone acknowledge
always @(posedge wb_clk_i or posedge wb_rst_i) begin
   if (wb_rst_i)
     wb_ack_o <= #1 1'b0;
   else
     wb_ack_o <= #1 wb_cyc_i & wb_stb_i & ~wb_ack_o;
end

//Wishbone termination Error
assign wb_err_o = 1'b0;

//Wishbone Interrupt Request
always @(posedge wb_clk_i or posedge wb_rst_i) begin
   if (wb_rst_i)
     wb_int_o <= #1 1'b0;
   else if (ie && tip && last_bit && pos_edge)
```

```
    wb_int_o <= #1 1'b1;
  else if (wb_ack_o)
    wb_int_o <= #1 1'b0;
end

//Divider Register
always @(posedge wb_clk_i or posedge wb_rst_i) begin
  if (wb_rst_i)
    divider <= #1 {`SPI_DIVIDER_LEN{1'b0}};
  else if (spi_divider_sel && wb_we_i && !tip) begin
  `ifdef SPI_DIVIDER_LEN_8
    if(wb_sel_i[0])
      divider <= #1 wb_dat_i[`SPI_DIVIDER_LEN-1:0];
  `endif
  `ifdef SPI_DIVIDER_LEN_16
    if(wb_sel_i[0])
      divider[7:0] <= #1 wb_dat_i[7:0];
    if(wb_sel_i[1])
      divider[`SPI_DIVIDER_LEN-1:8] <= #1 wb_dat_i[
        `SPI_DIVIDER_LEN-1:8];
  `endif
  `ifdef SPI_DIVIDER_LEN_24
    if(wb_sel_i[0])
      divider[7:0] <= #1 wb_dat_i[7:0];
    if(wb_sel_i[1])
      divider[15:8] <= #1 wb_dat_i[15:8];
    if(wb_sel_i[2])
      divider[`SPI_DIVIDER_LEN-1:16] <= #1 wb_dat_i[
        `SPI_DIVIDER_LEN-1:16];
  `endif
  `ifdef SPI_DIVIDER_LEN_32
    if(wb_sel_i[0])
      divider[7:0] <= #1 wb_dat_i[7:0];
    if(wb_sel_i[1])
```

```
      divider[15:8] <= #1 wb_dat_i[15:8];
    if(wb_sel_i[2])
      divider[23:16] <= #1 wb_dat_i[23:16];
    if(wb_sel_i[3])
      divider[`SPI_DIVIDER_LEN-1:24] <= #1 wb_dat_i[
         `SPI_DIVIDER_LEN-1:24];
  `endif
  end
end


//Control and Status Register
always @(posedge wb_clk_i or posedge wb_rst_i) begin
  if (wb_rst_i)
    ctrl <= #1 {`SPI_CTRL_BIT_NB{1'b0}};
  else if (spi_ctrl_sel && wb_we_i && !tip) begin
    if (wb_sel_i[0])
      ctrl[7:0] <= #1 wb_dat_i[7:0] | {7'b0, ctrl[0]};
    if (wb_sel_i[1])
      ctrl[`SPI_CTRL_BIT_NB-1:8] <= #1 wb_dat_i[`SPI_CTRL_BIT_NB
         -1:8];
  end
  else if (tip && last_bit && pos_edge)
    ctrl[`SPI_CTRL_GO] <= #1 1'b0;
end

assign rx_negedge = ctrl[`SPI_CTRL_RX_NEGEDGE];
assign tx_negedge = ctrl[`SPI_CTRL_TX_NEGEDGE];
assign go = ctrl[`SPI_CTRL_GO];
assign char_len = ctrl[`SPI_CTRL_CHAR_LEN];
assign lsb = ctrl[`SPI_CTRL_LSB];
assign ie = ctrl[`SPI_CTRL_IE];
assign ass = ctrl[`SPI_CTRL_ASS];
```

```
//Slave Select Register
always @(posedge wb_clk_i or posedge wb_rst_i) begin
  if (wb_rst_i)
    ss <= #1 {`SPI_SS_NB{1'b0}};
  else if (spi_ss_sel && wb_we_i && !tip) begin
    `ifdef SPI_SS_NB_8
     if(wb_sel_i[0])
       ss <= #1 wb_dat_i[`SPI_SS_NB-1:0];
    `endif
    `ifdef SPI_SS_NB_16
     if(wb_sel_i[0])
       ss[7:0] <= #1 wb_dat_i[7:0];
     if(wb_sel_i[1])
       ss[`SPI_SS_NB-1:8] <= #1 wb_dat_i[`SPI_SS_NB-1:8];
    `endif
    `ifdef SPI_SS_24
     if(wb_sel_i[0])
       ss[7:0] <= #1 wb_dat_i[7:0];
     if(wb_sel_i[1])
       ss[15:8] <= #1 wb_dat_i[15:8];
     if(wb_sel_i[2])
       ss[`SPI_SS_NB-1:16] <= #1 wb_dat_i[`SPI_SS_NB-1:16];
    `endif
    `ifdef SPI_SS_NB_32
     if(wb_sel_i[0])
       ss[7:0] <= #1 wb_dat_i[7:0];
     if(wb_sel_i[1])
       ss[15:8] <= #1 wb_dat_i[15:8];
     if(wb_sel_i[2])
       ss[23:16] <= #1 wb_dat_i[23:16];
     if(wb_sel_i[3])
       ss[`SPI_SS_NB-1:24] <= #1 wb_dat_i[`SPI_SS_NB-1:24];
    `endif
  end
end

assign ss_o = ~((ss & {`SPI_SS_NB{tip & ass}}) | (ss & {
   `SPI_SS_NB{!ass}}));

as_spi_clkgen Clock_Gen (
        .rst(wb_rst_i),
        .in_clk(wb_clk_i),
                     .in_clk_en(tip),
                     .go(go),
                     .last_clk(last_bit),
                     .clk_divider(divider),
                     .out_clk(sclk_o),
                     .out_clk_pos_edge(pos_edge),
                     .out_clk_neg_edge(neg_edge)
                     );

as_spi_shift  SHIFT_REGISTER
                     (
        .rst(wb_rst_i),
        .clk(wb_clk_i),
                     .latch(spi_tx_sel[3:0] & {4{wb_we_i}}),
                     .byte_sel(wb_sel_i),
                     .len(char_len[`SPI_CHAR_LEN_BITS-1:0]),
                     .lsb(lsb),
                     .go(go),
                     .pos_edge(pos_edge),
                     .neg_edge(neg_edge),
                     .rx_negedge(rx_negedge),
                     .tx_negedge(tx_negedge),
                     .tip(tip),
              .last(last_bit),
                     .p_in(wb_dat_i),
                     .p_out(rx),
                     .s_clk(sclk_o),
                     .s_in(miso_i),
                     .s_out(mosi_o)
                     );


endmodule // as_spi
```

I.9.4 SPI Slave

```
`include "src/timescale.v"

module as_spi_slave
        (
        input rst,
                   //Reset
        input ss,
                   //Slave Select
        input sclk,
                   //Serial Clock
        input mosi,
                   //Master Out Slave In
        output reg miso                                  //Master
           In Slave Out
        );


        reg rx_negedge;                                  //Data
           Received on negedge
        reg tx_negedge;
           //Data transmitted on negedge
        reg [127:0] data;
           //Data register

        always @(posedge(sclk && !rx_negedge) or negedge(sclk &&
           rx_negedge) or rst) begin
          if (rst)
                 data <= #1 128'b0;
          else if (!ss)
              data <= #1 {data[126:0],mosi};
```

```
  end

      always @(posedge(sclk && !tx_negedge) or negedge(sclk &&
        tx_negedge)) begin
        miso <= #1 data[127];
      end

endmodule
```